# Polyspace® Bug Finder™

## Reference

R2023a

MathWorks®

## How to Contact MathWorks

| | |
|---|---|
| Latest news: | `www.mathworks.com` |
| Sales and services: | `www.mathworks.com/sales_and_services` |
| User community: | `www.mathworks.com/matlabcentral` |
| Technical support: | `www.mathworks.com/support/contact_us` |
| Phone: | 508-647-7000 |

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

*Polyspace® Bug Finder™ Reference*

#### Trademarks

#### Patents

**Revision History**

| | | |
|---|---|---|
| September 2013 | Online Only | New for Version 1.0 (Release 2013b) |
| March 2014 | Online Only | Revised for Version 1.1 (Release 2014a) |
| October 2014 | Online Only | Revised for Version 1.2 (Release 2014b) |
| March 2015 | Online Only | Revised for Version 1.3 (Release 2015a) |
| September 2015 | Online Only | Revised for Version 2.0 (Release 2015b) |
| October 2015 | Online Only | Rereleased for Version 1.3.1 (Release 2015aSP1) |
| March 2016 | Online Only | Revised for Version 2.1 (Release 2016a) |
| September 2016 | Online Only | Revised for Version 2.2 (Release 2016b) |
| March 2017 | Online Only | Revised for Version 2.3 (Release 2017a) |
| September 2017 | Online Only | Revised for Version 2.4 (Release 2017b) |
| March 2018 | Online Only | Revised for Version 2.5 (Release 2018a) |
| September 2018 | Online Only | Revised for Version 2.6 (Release 2018b) |
| March 2019 | Online Only | Revised for Version 3.0 (Release 2019a) |
| September 2019 | Online Only | Revised for Version 3.1 (Release 2019b) |
| March 2020 | Online Only | Revised for Version 3.2 (Release 2020a) |
| September 2020 | Online Only | Revised for Version 3.3 (Release 2020b) |
| March 2021 | Online Only | Revised for Version 3.4 (Release 2021a) |
| September 2021 | Online Only | Revised for Polyspace Bug Finder Version 3.5, Polyspace Bug Finder Server Version 3.5, and Polyspace Bug Finder Access Version 3.1 (Release 2021b) |
| March 2022 | Online Only | Revised for Polyspace Bug Finder Version 3.6, Polyspace Bug Finder Server Version 3.6, and Polyspace Access Version 4.0 (Release 2022a) |
| September 2022 | Online Only | Revised for Polyspace Bug Finder Version 3.7, Polyspace Bug Finder Server Version 3.7, and Polyspace Access Version 4.1 (Release 2022b) |
| March 2023 | Online Only | Revised for Polyspace Bug Finder Version 3.8, Polyspace Bug Finder Server Version 3.8, and Polyspace Access Version 4.2 (Release 2023a) |

# Contents

## 1 Introduction

## Polyspace Analysis Options

## 2 Analysis Options

## 3 Analysis Options, Command-Line Only

## Polyspace DOS/UNIX Commands

## 4 Polyspace DOS/Unix Commands

## MATLAB and Simulink Functions, Classes, and Methods

## 5 Functions, Properties, Classes, and Apps

## 6 Configuration Parameters

## Polyspace Results: Defect Checkers

Polyspace Results: Coding Standards

## Polyspace Results: Code Metrics

## Polyspace Reports Components

## Polyspace Bug Finder Assumptions

1

# Introduction

# About This Reference

This Reference covers all Polyspace Bug Finder products:

- Polyspace Bug Finder
- Polyspace Bug Finder Server™
- Polyspace Access™

Depending on how you set up a Bug Finder run, you might be running an analysis from one of these locations:

- **Desktop**: If you are running an analysis and reviewing the results on your desktop, you use Polyspace Bug Finder. More specifically, you use the Polyspace user interface or the `polyspace-bug-finder` command to run an analysis.
- **Server**: If you are running an analysis on a server, or reviewing the results from a server run on a web browse, you use:
  - Polyspace Bug Finder Server, more specifically, the `polyspace-bug-finder-server` command, to run the analysis.
  - Polyspace Access to host the analysis results (for review on a web browser).
- **IDE**: If you are running an analysis on the current file in your Integration Development Environment (IDE), you use Polyspace as You Code in your IDEs (or the `polyspace-bug-finder-access` command). Polyspace as You Code is a feature available with Polyspace Access.

Whatever your platform, the Bug Finder analysis engine underlies all Bug Finder products. In particular, most analysis options, commands and result types are common to all three platforms.

# Polyspace Analysis Options

2

# Analysis Options

# Source code language (-lang)

Specify language of source files

## Description

Specify the language of your source files. Before specifying other configuration options, choose this option because other options change depending on your language selection. This option applies to both dynamic testing and static analysis.

If you add files during project setup, the language selection can change from the default.

| Files Added | Source Code Language |
|---|---|
| Only files with extension `.c` | `C` |
| Only files with extension `.cpp` or `.cc` | `CPP` |
| Files with extension `.c`, `.cpp`, and `.cc` | `C-CPP` |

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 2-2 for ways in which the source code language can be automatically determined.

**Command line and options file**: Use the option `-lang`. See “Command-Line Information” on page 2-3.

## Settings

**Default:** Based on file extensions.

`C`

If your project contains only C files, choose this setting. This value restricts the verification to C language conventions. All files are interpreted as C files, regardless of their file extension.

`CPP`

If your project contains only C++ files, choose this setting. This value restricts the verification to C++ language conventions. All files are interpreted as C++ files, regardless of their file extension.

`C-CPP`

If your project contains C and C++ source files, choose this setting. This value allows for C and C ++ language conventions. `.c` files are interpreted as C files. Other file extensions are interpreted as C++ files.

## Dependencies

- The language option allows and disallows many options and option values. Some options change depending on your language selection. For more information, see the individual analysis option pages.

- If you create a Polyspace project or options file from your build system using the `polyspace-configure` command or `polyspaceConfigure` function, the value of this option is determined by the file extensions.

  For a project with both `.c` and `.cpp` files, the language option `C-CPP` is used. During the analysis, each file is compiled based on the language standard determined by the file extensions. After the compilation, Polyspace verifies such mixed projects as C++ projects.

## Tips

For a project with both `.c` and `.cpp` files, if you use the language `C-CPP`, each file is compiled based on the language standard determined by the file extensions. After the compilation, Polyspace verifies such mixed projects as C++ projects.

In particular, the analysis links all files as C++ files. Because of differences in linking behavior between C and C++, you might see differences in linking errors or warnings when C files in the mixed C-C++ projects are compiled with language `C-CPP` versus language `C`.

## Command-Line Information

**Parameter:** `-lang`
**Value:** `c | cpp| c-cpp`
**Default:** Based on file extensions
**Example (Bug Finder):** `polyspace-bug-finder -lang c-cpp -sources "`*`file1.c,file2.cpp`*`"`
**Example (Code Prover):** `polyspace-code-prover -lang cpp -sources "`*`file1.cpp,file2.cpp`*`"`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -lang c-cpp -sources "`*`file1.c,file2.cpp`*`"`
**Example (Code Prover Server):** `polyspace-code-prover-server -lang cpp -sources "`*`file1.cpp,file2.cpp`*`"`
**Example (Bug Finder):** `polyspace-bug-finder -lang c -sources "`*`file1.c,file2.c`*`"`
**Example (Code Prover):** `polyspace-code-prover -lang c -sources "`*`file1.c,file2.c`*`"`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -lang c -sources "`*`file1.c,file2.c`*`"`
**Example (Code Prover Server):** `polyspace-code-prover-server -lang c -sources "`*`file1.c,file2.c`*`"`

## See Also

`C standard version (-c-version)` | `C++ standard version (-cpp-version)`

# C standard version (`-c-version`)

Specify C language standard followed in source code

## Description

Specify the C language standard that you follow in your source code. This option applies to both dynamic testing and static analysis.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 2-5 for other options that you must enable.

**Command line and options file**: Use the option `-c-version`. See “Command-Line Information” on page 2-5.

### Why Use This Option

Use this option so that Polyspace can allow features specific to a C standard version during compilation. For instance, if you compile with GCC using the flag `-ansi` or `-std=c90`, specify `c90` for this option. If you are not sure of the language standard, specify `defined-by-compiler`.

For instance, suppose you use the boolean data type `_Bool` in your code. This type is defined in the C99 standard but unknown in prior standards such as C90. If the Polyspace compilation follows the C90 standard, you can see compilation errors.

Some MISRA C™ rules are different based on whether you use the C90 or C99 standard. For instance, MISRA C C:2012 Rule 5.2 requires that identifiers in the same scope and name space shall be distinct. If you use the C90 standard, different identifiers that have the same first 31 characters violate this rule. If you use the C99 standard, the number of characters increase to 63.

## Settings

**Default:** `defined-by-compiler`

`defined-by-compiler`

The analysis uses a standard based on your specification for `Compiler (-compiler)`.

See “C/C++ Language Standard Used in Polyspace Analysis”.

`c90`

The analysis uses the C90 Standard (ISO®/IEC 9899:1990).

`c99`

The analysis uses the C99 Standard (ISO/IEC 9899:1999).

`c11`

The analysis uses the C11 Standard (ISO/IEC 9899:2011).

See also “C11 Language Elements Supported in Polyspace”.

`c17`

The analysis uses the C17 Standard (ISO/IEC 9899:2018).

This version addresses defects in C11 Standard but does not introduce new language features. The value of the `__STDC_VERSION__` macro is increased to `201710L`.

## Dependencies

- This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.
- If you create a project or options file from your build system using the `polyspace-configure` command or `polyspaceConfigure` function, the value of this option is automatically determined from your build system.

  If the build system uses different standards for different files, the subsequent Polyspace analysis can emulate your build system and use different standards for compiling those files. If you open such a project in the Polyspace user interface, the option value is shown as `defined-by-compiler`. However, instead of one standard, Polyspace uses the hidden option `-options-for-sources` to associate different standards with different files.

## Command-Line Information

**Parameter:** `-c-version`
**Value:** `defined-by-compiler` | `c90` | `c99` | `c11` | `c17`
**Default:** `defined-by-compiler`
**Example (Bug Finder):** `polyspace-bug-finder -lang c -sources "`*`file1.c,file2.c`*`" -c-version c90`
**Example (Code Prover):** `polyspace-code-prover -lang c -sources "`*`file1.c,file2.c`*`" -c-version c90`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -lang c -sources "`*`file1.c,file2.c`*`" -c-version c90`
**Example (Code Prover Server):** `polyspace-code-prover-server -lang c -sources "`*`file1.c,file2.c`*`" -c-version c90`

## See Also

`Source code language (-lang)` | `C++ standard version (-cpp-version)`

### Topics

“Specify Polyspace Analysis Options”
“C/C++ Language Standard Used in Polyspace Analysis”
“C11 Language Elements Supported in Polyspace”

# C++ standard version (-cpp-version)

Specify C++ language standard followed in source code

## Description

Specify the C++ language standard that you follow in your source code. This option applies to both dynamic testing and static analysis.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See "Dependencies" on page 2-7 for other options that you must enable.

**Command line and options file**: Use the option -cpp-version. See "Command-Line Information" on page 2-7.

### Why Use This Option

Use this option so that Polyspace can allow features from a specific version of the C++ language standard during compilation. For instance, if you compile with GCC using the flag -std=c++11 or -std=gnu++11, specify cpp11 for this option. If you are not sure of the language standard, specify defined-by-compiler.

For instance, suppose you use range-based for loops. This type of for loop is defined in the C++11 standard but unrecognized in prior standards such as C++03. If the Polyspace compilation uses the C ++03 standard, you can see compilation errors.

To check if your compiler allows features specific to a standard, compile code with macros specific to the standard using compiler settings that you typically use. For instance, to check for C++11-specific features, compile this code. The code contains a C++11-specific keyword nullptr. If the macro __cplusplus is not 201103L (indicating C++11), this keyword is used and causes a compilation error.

```
#if defined(__cplusplus) && __cplusplus >= 201103L
    /* C++11 compiler */
#else
    void* ptr = nullptr;
#endif
```

If the code compiles, use cpp11 for this option.

## Settings

**Default:** defined-by-compiler

defined-by-compiler

The analysis uses a standard based on your specification for Compiler (-compiler).

See "C/C++ Language Standard Used in Polyspace Analysis".

`cpp03`

The analysis uses the C++03 Standard (ISO/IEC 14882:2003).

`cpp11`

The analysis uses the C++11 Standard (ISO/IEC 14882:2011).

See also “C++11 Language Elements Supported in Polyspace”.

`cpp14`

The analysis uses the C++14 Standard (ISO/IEC 14882:2014).

See also “C++14 Language Elements Supported in Polyspace”.

`cpp17`

The analysis uses the C++17 Standard (ISO/IEC 14882:2017).

See also “C++17 Language Elements Supported in Polyspace”.

## Dependencies

- This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.
- If you create a project or options file from your build system using the `polyspace-configure` command or `polyspaceConfigure` function, the value of this option is automatically determined from your build system.

  If the build system uses different standards for different files, the subsequent Polyspace analysis can emulate your build system and use different standards for compiling those files. If you open such a project in the Polyspace user interface, the option value is shown as `defined-by-compiler`. However, instead of one standard, Polyspace uses multiple standards for compiling the files. The analysis uses the hidden option `-options-for-sources` to associate different standards with different files.

## Command-Line Information

**Parameter:** `-cpp-version`
**Value:** `defined-by-compiler` | `cpp03` | `cpp11` | `cpp14` | `cpp17`
**Default:** `defined-by-compiler`
**Example (Bug Finder):** `polyspace-bug-finder -lang c -sources "`*`file1.c,file2.c`*`" -cpp-version cpp11`
**Example (Code Prover):** `polyspace-code-prover -lang c -sources "`*`file1.c,file2.c`*`" -cpp-version cpp11`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -lang c -sources "`*`file1.c,file2.c`*`" -cpp-version cpp11`
**Example (Code Prover Server):** `polyspace-code-prover-server -lang c -sources "`*`file1.c,file2.c`*`" -cpp-version cpp11`

## See Also

`Source code language (-lang)` | `C standard version (-c-version)`

### Topics

“Specify Polyspace Analysis Options”
“C/C++ Language Standard Used in Polyspace Analysis”

"C++11 Language Elements Supported in Polyspace"
"C++14 Language Elements Supported in Polyspace"
"C++17 Language Elements Supported in Polyspace"

# Compiler (-compiler)

Specify the compiler that you use to build your source code

## Description

Specify the compiler that you use to build your source code.

Polyspace fully supports the most common compilers used to develop embedded applications. See the list below. For these compilers, you can run analysis simply by specifying your compiler and target processor. For other compilers, specify `generic` as compiler name. If you face compilation errors, explicitly define compiler-specific extensions to work around the errors.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

**Command line and options file**: Use the option `-compiler`. See “Command-Line Information” on page 2-17.

### Why Use This Option

Polyspace uses this information to interpret syntax that is not part of the C/C++ Standard, but comes from language extensions.

For example, the option allows additional language keywords, such as `sfr`, `sbit`, and `bit`. If you do not specify your compiler, these additional keywords can cause compilation errors during Polyspace analysis.

Polyspace does not actually invoke your compiler for compilation. In particular:

- You cannot specify compiler flags directly in the Polyspace analysis. To emulate your compiler flags, trace your build command or manually specify equivalent Polyspace analysis options. See “Specify Target Environment and Compiler Behavior”.
- Code Prover has a linking policy that is stricter than regular compilers. For instance, if your compiler allows declaration mismatches with specific compiler options, you cannot emulate this linking policy in Code Prover. See “Troubleshoot Compilation and Linking Errors” (Polyspace Code Prover).

## Settings

**Default:** `generic`

### GCC Compilers

`gnu3.4`

Analysis allows GCC 3.4 syntax.

`gnu4.6`

Analysis allows GCC 4.6 syntax.

`gnu4.7`

Analysis allows GCC 4.7 syntax.

For unsupported GCC extensions, see “Limitations” on page 2-15.

`gnu4.8`

Analysis allows GCC 4.8 syntax.

For unsupported GCC extensions, see “Limitations” on page 2-15.

`gnu4.9`

Analysis allows GCC 4.9 syntax.

For unsupported GCC extensions, see “Limitations” on page 2-15.

`gnu5.x`

Analysis allows GCC 5.x syntax. For a list of available GCC 5.x releases, see GCC releases.

If you select `gnu5.x`, you can specify only a subset of GCC based compiler targets by using the option `Target processor type (-target)`. To specify other targets that are not part of this subset, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 2-15.

`gnu6.x`

Analysis allows GCC 6.x syntax. For a list of available GCC 6.x releases, see GCC releases.

If you select `gnu6.x`, you can specify only a subset of GCC based compiler targets by using the option `Target processor type (-target)`. To specify other targets that are not part of this subset, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 2-15.

`gnu7.x`

Analysis allows GCC 7.x syntax. For a list of available GCC 7.x releases, see GCC releases.

If you select `gnu7.x`, you can specify only a subset of GCC based compiler targets by using the option `Target processor type (-target)`. To specify other targets that are not part of this subset, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 2-15.

`gnu8.x`

Analysis allows GCC 8.x syntax. For a list of available GCC 8.x releases, see GCC releases.

If you select `gnu8.x`, you can specify only a subset of GCC based compiler targets by using the option `Target processor type (-target)`. To specify other targets that are not part of this subset, use the option `Generic target options`..

For unsupported GCC extensions, see “Limitations” on page 2-15.

`gnu9.x`

Analysis allows GCC 9.x syntax. For a list of available GCC 9.x releases, see GCC releases.

If you select `gnu9.x`, you can specify only a subset of GCC based compiler targets by using the option `Target processor type (-target)`. To specify other targets that are not part of this subset, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 2-15.

`gnu10.x`

Analysis allows GCC 10.x syntax. For a list of available GCC 10.x releases, see GCC releases.

If you select `gnu10.x`, you can specify only a subset of GCC based compiler targets by using the option `Target processor type (-target)`. To specify other targets that are not part of this subset, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 2-15.

`gnu11.x`

Analysis allows GCC 11.x syntax. For a list of available GCC 11.x releases, see GCC releases.

If you select `gnu11.x`, you can specify only a subset of GCC based compiler targets by using the option `Target processor type (-target)`. To specify other targets that are not part of this subset, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 2-15.

`gnu12.x`

Analysis allows GCC 12.x syntax. For a list of available GCC 12.x releases, see GCC releases.

If you select `gnu12.x`, you can specify only a subset of GCC based compiler targets by using the option `Target processor type (-target)`. To specify other targets that are not part of this subset, use the option `Generic target options`.

For unsupported GCC extensions, see “Limitations” on page 2-15.

### Clang Compilers

`clang3.x`

Analysis allows Clang syntax for these versions:

- 3.5.0, 3.5.1, and 3.5.2
- 3.6.0, 3.6.1, and 3.6.2
- 3.7.0 and 3.7.1
- 3.8.0 and 3.8.1
- 3.9.0 and 3.9.1

`clang4.x`

Analysis allows Clang 4.0.0, and 4.0.1 syntax.

`clang5.x`

Analysis allows Clang 5.0.0, 5.0.1, and 5.0.2 syntax.

`clang6.x`

Analysis allows Clang 6.0.0 and 6.0.1 syntax.

`clang7.x`

Analysis allows Clang 7.0.0, 7.0.1, and 7.1.0 syntax.

`clang8.x`

Analysis allows Clang 8.0.0 and 8.0.1 syntax.

`clang9.x`

Analysis allows Clang 9.0.0 and 9.0.1 syntax.

`clang10.x`

Analysis allows Clang 10.0.0 and 10.0.1 syntax.

`clang11.x`

Analysis allows Clang 11.0.0, 11.0.1, and 11.1.0 syntax.

`clang12.x`

Analysis allows Clang 12.0.0 and 12.0.1 syntax.

`clang13.x`

Analysis allows Clang 13.0.0 and 13.0.1 syntax.

#### Visual Studio Compilers

`visual9.0`

Analysis allows Microsoft® Visual C++® 2008 syntax.

`visual10.0`

Analysis allows Microsoft Visual C++ 2010 syntax.

`visual11.0`

Analysis allows Microsoft Visual C++ 2012 syntax.

`visual12.0`

Analysis allows Microsoft Visual C++ 2013 syntax.

`visual14.0`

Analysis allows Microsoft Visual C++ 2015 syntax (supports Microsoft Visual Studio® update 2).

`visual15.x`

Analysis allows Microsoft Visual C++ 2017 syntax. For a list of available Microsoft Visual Studio 2017 versions, see Visual Studio 2017 Release Notes History.

`visual16.x`

Analysis allows Microsoft Visual C++ 2019 syntax. For a list of available Microsoft Visual Studio 2019 versions, see Visual Studio 2019 Release Notes History.

#### Other Compilers

`armcc`

Analysis allows non-ANSI® C syntax and semantics associated with the ARM® v5 compiler.

If you select `armcc`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the ARM v5 compiler.

See `ARM v5 Compiler (-compiler armcc)`.

`armclang`

Analysis allows non-ANSI C syntax and semantics associated with the ARM v6 compiler.

If you select `armclang`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the ARM v6 compiler.

See `ARM v6 Compiler (-compiler armclang)`.

`codewarrior`

Analysis allows non-ANSI C syntax and semantics associated with the NXP CodeWarrior® compiler.

If you select `codewarrior`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the NXP CodeWarrior compiler.

See `NXP CodeWarrior Compiler (-compiler codewarrior)`.

`cosmic`

Analysis allows non-ANSI C syntax and semantics associated with the Cosmic compiler.

If you select `cosmic`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the Comic compiler.

See `Cosmic Compiler (-compiler cosmic)`.

`diab`

Analysis allows non-ANSI C syntax and semantics associated with the Wind River® Diab compiler.

If you select `diab`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the Wind River Diab compiler.

See `Diab Compiler (-compiler diab)`.

`generic`

Analysis allows only standard syntax.

The language standard is determined by your choice for the following options:

- `C standard version (-c-version)`
- `C++ standard version (-cpp-version)`

If you do not specify a standard explicitly, the standard depends on your choice of compiler.

`greenhills`

Analysis allows non-ANSI C syntax and semantics associated with a Green Hills® compiler.

If you select `greenhills`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for a Green Hills compiler.

See `Green Hills Compiler (-compiler greenhills)`.

`iar`

Analysis allows non-ANSI C syntax and semantics associated with the compilers from IAR Systems (www.iar.com).

`iar-ew`

Analysis allows non-ANSI C syntax and semantics associated with the IAR Embedded Workbench compiler.

If you select `iar-ew`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the IAR Embedded Workbench compiler.

See `IAR Embedded Workbench Compiler (-compiler iar-ew)`.

`intel`

Analysis allows non-ANSI C syntax and semantics associated with the Intel® C++ Compiler Classic (icc/icl) compiler.

If you select `intel`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the Intel C++ Compiler Classic (icc/icl) compiler.

See `Intel C++ Compiler Classic (icc/icl) (-compiler intel)`.

`keil`

Analysis allows non-ANSI C syntax and semantics associated with the Keil products from ARM (www.keil.com).

`microchip`

Analysis allows non-ANSI C syntax and semantics associated with the MPLAB XC8 C compiler.

If you select `microchip`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the MPLAB XC8 C compiler.

See `MPLAB XC8 C Compiler (-compiler microchip)`.

`renesas`

Analysis allows non-ANSI C syntax and semantics associated with the Renesas® compiler.

If you select `renesas`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the Renesas compiler.

See `Renesas Compiler (-compiler renesas)`.

`tasking`

Analysis allows non-ANSI C syntax and semantics associated with the TASKING compiler.

If you select `tasking`,in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the TASKING compiler.

See `TASKING Compiler (-compiler tasking)`.

`ti`

Analysis allows non-ANSI C syntax and semantics associated with the Texas Instruments® compiler.

If you select `ti`, in the user interface of the Polyspace desktop products, the option `Target processor type (-target)` shows only the targets that are allowed for the Texas Instruments compiler.

See `Texas Instruments Compiler (-compiler ti)`.

## Tips

- Your compiler specification determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.
  - To override the macro definition, use the option `Preprocessor definitions (-D)`.
  - To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.
- If you use a Visual Studio compiler, you must use a `Target processor type (-target)` option that sets `long long` to 64 bits. Compatible targets include: `i386`, `sparc`, `m68k`, `powerpc`, `tms320c3x`, `sharc21x61`, `mpc5xx`, `x86_64`, or `mcpu` with `long long` set to 64 (`-long-long-is-64bits` at the command line).
- If you use the option `Check JSF AV C++ rules (-jsf-coding-rules)`, select the compiler `generic`. If you use another compiler, Polyspace cannot check the JSF® coding rules that require conforming to the ISO standard. For example, AV Rule 8: "All code shall conform to ISO/IEC 14882:2002(E) standard C++."

## Limitations

### GNU Compilers

Polyspace does not support certain features of GNU compilers:

- **GNU® compilers versions 4.7 and later:**
  - Nested functions.

    For instance, the function `bar` is nested in function `foo`:

    ```
    int foo (int a, int b)
    {
      int bar (int c) { return c * c; }

      return bar (a) + bar (b);
    }
    ```
  - Binary operations with vector types where one operand uses the shorthand notation for uniform vectors.

    For instance, in the addition operation, `2+a`, 2 is used as a shorthand notation for {2,2,2,2}.

    ```
    typedef int v4si __attribute__ ((vector_size (16)));
    v4si res, a = {1,2,3,4};

    res = 2 + a;  /* means {2,2,2,2} + a  */
    ```
  - Forward declaration of function parameters.

    For instance, the parameter `len` is forward declared:

    ```
    void func (int len; char data[len][len], int len)
    {
      /* … */
    }
    ```
  - Complex integer data types.

However, complex floating point data types are supported.

- Initialization of structures with flexible array members using an initialization list.

  For instance, the structure S has a flexible array member tab. A variable of type S is directly initialized with an initialization list.

  ```
  struct S {
      int x;
      int tab[];            /* flexible array member - not supported */
  };
  struct S s = { 0, 1, 2} ;
  ```

  You see a warning during analysis and a red check in the results when you dereference, for instance, s.tab[1].
- 128-bit variables.

  Polyspace cannot analyze this data type semantically. Bug Finder allows use of 128-bit data types, but Code Prover shows a compilation error if you use such a data type, for instance, the GCC extension __float128.

- **GNU compilers version 7.x:**

  - Type names _FloatN and _FloatNx are not semantically supported. The analysis treats them as type float, double, or long double.
  - Constants of type _FloatN or _FloatNx with suffixes fN, FN, or fNx, such as 1.2f123 or 2.3F64x are not supported.

### Visual Studio Compilers

Polyspace does not support certain features of Visual Studio compilers:

- C++ Accelerated Massive Parallelism (AMP).

  C++ AMP is a Visual Studio feature that accelerates your C++ code execution for certain types of data-parallel hardware on specific targets. You typically use the restrict keyword to enable this feature.

  ```
  void Buffer() restrict(amp)
  {
    ...
  }
  ```
- __assume statements.

  You typically use __assume with a condition that is false. The statement indicates that the optimizer must assume the condition to be henceforth true. Code Prover cannot reconcile this contradiction. You get the error:

  ```
  Asked for compulsory presence of absent entity : assert
  ```
- Managed Extensions for C++ (required for the .NET Framework), or its successor, C++/CLI (C++ modified for Common Language Infrastructure)
- __declspec keyword with attributes other than noreturn, nothrow, selectany or thread.

### Polyspace System Headers

If you do not specify the path to your compiler headers, Polyspace uses its own system headers and your project might not compile even if your code compiles with your compiler.

To make sure that Polyspace uses your compiler header files, run `polyspace-configure` or specify the paths to your compiler header files manually. See "Provide Standard Library Headers for Polyspace Analysis".

## Command-Line Information

**Parameter:** `-compiler`
**Value (GCC):** `gnu3.4 | gnu4.6 | gnu4.7 | gnu4.8 | gnu4.9 | gnu5.x | gnu6.x | gnu7.x | gnu8.x | gnu9.x | gnu10.x | gnu11.x | gnu12.x`
**Value (Clang):** `clang3.x | clang4.x | clang5.x | clang6.x | clang7.x | clang8.x | clang9.x | clang10.x | clang11.x | clang12.x | clang13.x`
**Value (Visual Studio):** `visual10.0 | visual11.0 | visual12.0 | visual14.0 | visual15.x | visual16.x | visual9.0`
**Value (Other):** `armcc | armclang | codewarrior | cosmic | diab | generic | greenhills | iar | iar-ew | intel | keil | microchip | renesas | tasking | ti`
**Default:** `generic`
**Example 1 (Bug Finder):** `polyspace-bug-finder -lang c -sources "`*`file1.c,file2.c`*`" -compiler gnu4.6`
**Example 2 (Bug Finder):** `polyspace-bug-finder -lang cpp -sources "`*`file1.cpp,file2.cpp`*`" -compiler visual9.0`
**Example 1 (Code Prover):** `polyspace-code-prover -lang c -sources "`*`file1.c,file2.c`*`" -lang c -compiler gnu4.6`
**Example 2 (Code Prover):** `polyspace-code-prover -lang cpp -sources "`*`file1.cpp,file2.cpp`*`" -compiler visual9.0`
**Example 1 (Bug Finder Server):** `polyspace-bug-finder-server -lang c -sources "`*`file1.c,file2.c`*`" -compiler gnu4.6`
**Example 2 (Bug Finder Server):** `polyspace-bug-finder-server -lang cpp -sources "`*`file1.cpp,file2.cpp`*`" -compiler visual9.0`
**Example 1 (Code Prover Server):** `polyspace-code-prover-server -lang c -sources "`*`file1.c,file2.c`*`" -lang c -compiler gnu4.6`
**Example 2 (Code Prover Server):** `polyspace-code-prover-server -lang cpp -sources "`*`file1.cpp,file2.cpp`*`" -compiler visual9.0`

## See Also

`Target processor type (-target)` | `C standard version (-c-version)` | `C++ standard version (-cpp-version)`

### Topics

"Specify Polyspace Analysis Options"
"Troubleshoot Compilation Errors"
"Specify Target Environment and Compiler Behavior"
"Supported Keil or IAR Language Extensions"

# Target processor type (`-target`)

Specify size of data types and endianness by selecting a predefined target processor

## Description

Specify the processor on which you deploy your code.

The target processor determines the sizes of fundamental data types and the endianness of the target machine. You can analyze code intended for an unlisted processor type by using one of the other processor types, if they share common data properties.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. To see the sizes of types, click the **Edit** button to the right of the **Target processor type** drop-down list.

For some compilers, in the user interface, you see only the processors allowed for that compiler. For these compilers, you also cannot see the data type sizes in the user interface. See the links in the table below for the data type sizes.

**Command line and options file**: Use the option `-target`. See “Command-Line Information” on page 2-20.

### Why Use This Option

You specify a target processor so that some of the Polyspace run-time checks are tailored to the data type sizes and other properties of that processor.

For instance, a variable can overflow for smaller values on a 32-bit processor such as i386 compared to a 64-bit processor such as x86_64. If you select x86_64 for your Polyspace analysis, but deploy your code to the i386 processor, your Polyspace results are not always applicable.

Once you select a target processor, you can specify if the default sign of char is signed or unsigned. To determine which signedness to specify, compile this code using the compiler settings that you typically use:

```
#include <limits.h>
int array[(char)UCHAR_MAX]; /* If char is signed, the array size is -1
```

If the code compiles, the default sign of char is unsigned. For instance, on a GCC compiler, the code compiles with the `-fsigned-char` flag and fails to compile with the `-funsigned-char` flag.

## Settings

**Default:** `i386`

This table shows the size of each fundamental data type that Polyspace considers. For some targets, you can modify the default size by clicking the **Edit** button to the right of the **Target processor type** drop-down list. The optional values for those targets are shown in [brackets] in the table.

| Target | char | short | int | long | long long | float | double | long double[a] | ptr | Default sign of char | endian | Alignment |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i386 | 8 | 16 | 32 | 32 | 64 | 32 | 64 | 96 | 32 | signed | Little | 32 |
| sparc | 8 | 16 | 32 | 32 | 64 | 32 | 64 | 128 | 32 | signed | Big | 64 |
| m68k[b] | 8 | 16 | 32 | 32 | 64 | 32 | 64 | 96 | 32 | signed | Big | 64 |
| powerpc | 8 | 16 | 32 | 32 | 64 | 32 | 64 | 128 | 32 | unsigned | Big | 64 |
| c-167 | 8 | 16 | 16 | 32 | 32 | 32 | 64 | 64 | 16 | signed | Little | 64 |
| tms320c3x | 32 | 32 | 32 | 32 | 64 | 32 | 32 | 64 | 32 | signed | Little | 32 |
| sharc21x61 | 32 | 32 | 32 | 32 | 64 | 32 | 32 [64] | 32 [64] | 32 | signed | Little | 32 |
| necv850 | 8 | 16 | 32 | 32 | 32 | 32 | 32 | 64 | 32 | signed | Little | 32 |
| hc08[c] | 8 | 16 | 16 [32] | 32 | 32 | 32 | 32 [64] | 32 [64] | 16[d] | unsigned | Big | 32 |
| hc12 | 8 | 16 | 16 [32] | 32 | 32 | 32 | 32 [64] | 32 [64] | 32[6] | signed | Big | 32 |
| mpc5xx | 8 | 16 | 32 | 32 | 64 | 32 | 32 [64] | 32 [64] | 32 | signed | Big | 32 |
| c18 | 8 | 16 | 16 | 32 [24][e] | 32 | 32 | 32 | 32 | 16 [24] | signed | Little | 8 |
| x86_64 | 8 | 16 | 32 | 64 [32][f] | 64 | 32 | 64 | 128 | 64 | signed | Little | 128 |
| mcpu... (Advanced)[g] | 8 [16] | 8 [16] | 16 [32] | 32 | 32 [64] | 32 | 32 [64] | 32 [64] | 16 [32] | signed | Little | 32 |
| Targets for ARM v5 compiler | See ARM v5 Compiler (-compiler armcc). | | | | | | | | | | | |
| Targets for ARM v6 compiler | See ARM v6 Compiler (-compiler armclang). | | | | | | | | | | | |
| Targets for NPX CodeWarrior compiler | See NXP CodeWarrior Compiler (-compiler codewarrior). | | | | | | | | | | | |
| Targets for Cosmic compiler | See Cosmic Compiler (-compiler cosmic). | | | | | | | | | | | |
| Targets for Diab compiler | See Diab Compiler (-compiler diab). | | | | | | | | | | | |
| Targets for Green Hills compiler | See Green Hills Compiler (-compiler greenhills). | | | | | | | | | | | |

| Target | cha<br>r | short | int | lon<br>g | long long | floa<br>t | double | long double[a] | ptr | Default sign of char | endian | Align ment |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Targets for IAR Embedded Workbench compiler | See `IAR Embedded Workbench Compiler (-compiler iar-ew)`. | | | | | | | | | | | |
| Targets for MPLAB XC8 C compiler | See `MPLAB XC8 C Compiler (-compiler microchip)` | | | | | | | | | | | |
| Targets for Renesas compiler | See `Renesas Compiler (-compiler renesas)`. | | | | | | | | | | | |
| Targets for TASKING compiler | See `TASKING Compiler (-compiler tasking)`. | | | | | | | | | | | |
| Targets for Texas Instruments compiler | See `Texas Instruments Compiler (-compiler ti)`. | | | | | | | | | | | |

[a] For targets where the size of `long double` is greater than 64 bits, the size used for computations is not always the same as the size listed in this table. The exceptions are:

- For targets `i386`, `x86_64` and `m68k`, 80 bits are used for computations, following the practice in common compilers.
- For the target `tms320c3x`, 40 bits are used for computation, following the TMS320C3x specifications.
- If you use a Visual compiler, the size of `long double` used for computations is the same as size of `double`, following the specification of Visual C++ compilers.

[b] The M68k family (68000, 68020, and so on) includes the “ColdFire” processor

[c] Non-ANSI C specified keywords and compiler implementation-dependent pragmas and interrupt facilities are not taken into account by this support

[d] All kinds of pointers (near or far pointer) have 2 bytes (hc08) or 4 bytes (hc12) of width physically.

[e] The `c18` target supports the type `short long` as 24 bits in size.

[f] Use option `-long-is-32bits` to support Microsoft C/C++ Win64 target.

[g] `mcpu` is a reconfigurable Micro Controller/Processor Unit target. You can use this type to configure one or more generic targets. For more information, see `Generic target options`.

## Tips

- If your processor is not listed, use a similar processor that shares the same characteristics, or create an `mcpu` generic target processor. See `Generic target options`.
- You can also create a custom target by explicitly stating sizes of fundamental types and so on with the option `-custom-target`.
- If your configuration uses both `-custom-target` and `-target` to specify targets, the analysis uses the target that you specify with `-custom-target`.

## Command-Line Information

**Parameter:** `-target`
**Value:** `i386 | sparc | m68k | powerpc | c-167 | tms320c3x | sharc21x61 | necv850 | hc08 | hc12 | mpc5xx | c18 | x86_64 | mcpu`
**Default:** `i386`

**Example (Bug Finder):** `polyspace-bug-finder -target m68k`
**Example (Code Prover):** `polyspace-code-prover -target m68k`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -target m68k`
**Example (Code Prover Server):** `polyspace-code-prover-server -target m68k`

You can override the default values for some targets by using specific command-line options. See the section **Command-Line Options** in `Generic target options`.

## See Also

### Polyspace Analysis Options

`-custom-target`

### Polyspace Results

`Lower Estimate of Size of Local Variables` | `Higher Estimate of Size of Local Variables`

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# ARM v5 Compiler (-compiler armcc)

Specify ARM v5 compiler

## Description

Specify armcc for the Compiler (-compiler) option if you compile your code with a ARM v5 compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select armcc for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a ARM v5 compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the armcc compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

## Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option -dump-preprocessing-info.

- To override the macro definition, use the option Preprocessor definitions (-D).
- To undefine a macro, use the option Disabled preprocessor definitions (-U).

## Command-Line Information

**Parameter:** -compiler armcc -target
**Value:** arm
**Default:** arm
**Example (Bug Finder):** polyspace-bug-finder -compiler armcc -target arm
**Example (Code Prover):** polyspace-code-prover -compiler armcc -target arm
**Example (Bug Finder Server):** polyspace-bug-finder-server -compiler armcc -target arm
**Example (Code Prover Server):** polyspace-code-prover-server -compiler armcc -target arm

## Tips

If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

# Version History

**Introduced in R2019a**

## See Also

`Compiler (-compiler)` | `Target processor type (-target)`

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# ARM v6 Compiler (`-compiler armclang`)

Specify ARM v6 compiler

## Description

Specify `armclang` for the `Compiler (-compiler)` option if you compile your code with a ARM v6 compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `armclang` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a ARM v6 compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the `armclang` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

## Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

## Command-Line Information

**Parameter:** `-compiler armclang -target`
**Value:** `arm | arm64`
**Default:** `arm`
**Example (Bug Finder):** `polyspace-bug-finder -compiler armclang -target arm64`
**Example (Code Prover):** `polyspace-code-prover -compiler armclang -target arm64`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -compiler armclang -target arm64`
**Example (Code Prover Server):** `polyspace-code-prover-server -compiler armclang -target arm64`

## Tips

If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

# Version History

**Introduced in R2019a**

## See Also

`Compiler (-compiler)` | `Target processor type (-target)`

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# NXP CodeWarrior Compiler (-compiler codewarrior)

Specify NXP CodeWarrior compiler

## Description

Specify `codewarrior` for `Compiler (-compiler)` if you compile your code using a NXP CodeWarrior compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `codewarrior` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a NXP CodeWarrior compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `codewarrior` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

## Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

## Command-Line Information

**Parameter:** `-compiler codewarrior -target`
**Value:** `s12z | powerpc`
**Default:** `s12z`
**Example (Bug Finder):** `polyspace-bug-finder -compiler codewarrior -target powerpc`
**Example (Code Prover):** `polyspace-code-prover -compiler codewarrior -target powerpc`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -compiler codewarrior -target powerpc`
**Example (Code Prover Server):** `polyspace-code-prover-server -compiler codewarrior -target powerpc`

## Tips

If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

## Version History

**Introduced in R2018a**

## See Also

`Compiler (-compiler)` | `Target processor type (-target)`

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# Cosmic Compiler (-compiler cosmic)

Specify Cosmic compiler

## Description

Specify `cosmic` for the `Compiler (-compiler)` option if you compile your code with a Cosmic compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `cosmic` for **Compiler**, in the user interface, you see only the processors allowed for a Cosmic compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the `cosmic` compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

## Settings

To see the default sizes in bits for the fundamental types that the target uses, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

## Command-Line Information

**Parameter:** `-compiler cosmic -target`
**Value:** `s12z`
**Default:** `s12z`
**Example (Bug Finder):** `polyspace-bug-finder -compiler cosmic -target s12z`
**Example (Code Prover):** `polyspace-code-prover -compiler cosmic -target s12z`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -compiler cosmic -target s12z`
**Example (Code Prover Server):** `polyspace-code-prover-server -compiler cosmic -target s12z`

## Tips

If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

# Version History

**Introduced in R2019b**

## See Also

`Compiler (-compiler)` | `Target processor type (-target)`

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# Diab Compiler (`-compiler diab`)

Specify the Wind River Diab compiler

## Description

Specify `diab` for `Compiler (-compiler)` if you compile your code using the Wind River Diab compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `diab` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for the Diab compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `diab` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

## Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

## Tips

- Polyspace does not support these Diab compiler features:
  - The preprocessor directives `#assert` and `#unassert`. Your code compiles but the software does not interpret these directives semantically.
  - Single-character constants in `#if` directives having the same value as the same character constant in the execution character set. Your code compiles but Polyspace does not consider that the character constants have the same value.
  - The extended `sizeof()` syntax using two arguments. For example, `sizeof(char, 2)`. Your code does not compile with Polyspace when you use this feature.
  - Statement expressions. For example, `({int y; y=foo(); y;})`. Your code does not compile with Polyspace when you use this feature.
  - The use of regular expressions with the `defined` preprocessor operator. For example `#if defined ("BSP_HW*")`. Your code does not compile with Polyspace when you use this feature.
- If you encounter errors during Polyspace analysis, see “Fix Polyspace Compilation Errors Related to Diab Compiler”.

- If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

### Command-Line Information

**Parameter:** `-compiler diab -target`
**Value:** `i386 | powerpc | arm | coldfire | mips | mcore | rh850 | superh | tricore`
**Default:** `powerpc`
**Example (Bug Finder):** `polyspace-bug-finder -compiler diab -target tricore`
**Example (Code Prover):** `polyspace-code-prover -compiler diab -target tricore`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -compiler diab -target tricore`
**Example (Code Prover Server):** `polyspace-code-prover-server -compiler diab -target tricore`

## Version History

**Introduced in R2016b**

### See Also

`Compiler (-compiler)` | `Target processor type (-target)`

#### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# Green Hills Compiler (-compiler greenhills)

Specify Green Hills compiler

## Description

Specify greenhills for Compiler (-compiler) if you compile your code using a Green Hills compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select greenhills for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a Green Hills compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the greenhills compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

## Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option -dump-preprocessing-info.

- To override the macro definition, use the option Preprocessor definitions (-D).
- To undefine a macro, use the option Disabled preprocessor definitions (-U).

## Tips

- If you encounter errors during a Polyspace analysis, see “Fix Polyspace Compilation Errors Related to Green Hills Compiler”
- Polyspace supports the embedded configuration for the i386 target. If your x86 Green Hills compiler is configured for native Windows® development, you can see compilation errors or incorrect analysis results with Code Prover. Contact Technical Support.

  For instance, Green Hills compilers consider a size of 12 bytes for long double for embedded targets, but 8 bytes for native Windows. Polyspace considers 12 bytes by default.
- If you create a Polyspace project from a build command that uses a Green Hills compiler, the compiler options -filetype and -os_dir are not implemented in the project. To emulate the -os_dir option, you can explicitly add the path argument of the option as an include folder to your Polyspace project.
- If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

## Command-Line Information

**Parameter:** `-compiler greenhills -target`
**Value:** `powerpc | powerpc64 | arm | arm64 | tricore | rh850 | arm | i386 | x86_64`
**Default:** `powerpc`
**Example (Bug Finder):** `polyspace-bug-finder -compiler greenhills -target arm`
**Example (Code Prover):** `polyspace-code-prover -compiler greenhills -target arm`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -compiler greenhills -target arm`
**Example (Code Prover Server):** `polyspace-code-prover-server -compiler greenhills -target arm`

# Version History

**Introduced in R2017b**

## See Also

`Compiler (-compiler)` | `Target processor type (-target)`

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# IAR Embedded Workbench Compiler (-compiler iar-ew)

Specify IAR Embedded Workbench compiler

## Description

Specify iar-ew for Compiler (-compiler) if you compile your code using a IAR Embedded Workbench compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select iar-ew for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a IAR Embedded Workbench compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the iar-ew compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

## Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option -dump-preprocessing-info.

- To override the macro definition, use the option Preprocessor definitions (-D).
- To undefine a macro, use the option Disabled preprocessor definitions (-U).

## Tips

- Polyspace does not support the use of $Super$$ and $Sub$$ to patch symbol definitions. For instance, the following code compiles correctly, but Polyspace considers that main() calls the extern func ($Super$$func) instead of the function func defined in this code ($Sub$$func):

```
/* void func() declared in another file */
extern void $Super$$func(int i);

int setup = 0;

void setup_func(int i) {
  setup = i;
}

/* this function should be called instead of the original extern func() */
void $Sub$$func(int i) {
  setup_foo(i);
  /* does some extra setup work */
  /* ... */
}

int main() {
  assert(setup = 0);
  func(1); // Should call $Sub$$func instead of $Super$$func
  assert(setup = 1);
  return 0;
}
```

To make sure that Polyspace calls the correct function when analyzing your code, replace all instance of `$Sub$$` with an empty string in all your files after preprocessing. See `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

- Polyspace does not support some constructs specific to the IAR compiler.

  For the list of unsupported constructs, see `codeprover_limitations.pdf` in *`polyspaceroot`*`\polyspace\verifier\code_prover_desktop`. Here, *`polyspaceroot`* is the MATLAB® installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.
- If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

## Command-Line Information

**Parameter:** `-compiler iar-ew -target`
**Value:** `arm | avr | msp430 | rh850 | riscv | rl78`
**Default:** `arm`
**Example (Bug Finder):** `polyspace-bug-finder -compiler iar-ew -target rl78`
**Example (Code Prover):** `polyspace-code-prover -compiler iar-ew -target rl78`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -compiler iar-ew -target rl78`
**Example (Code Prover Server):** `polyspace-code-prover-server -compiler iar-ew -target rl78`

# Version History

**Introduced in R2018a**

## See Also

`Compiler (-compiler)` | `Target processor type (-target)`

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# Intel C++ Compiler Classic (icc/icl) (-compiler intel)

Specify Intel C++ Compiler Classic (icc/icl)

## Description

Specify intel for the Compiler (-compiler) option if you compile your code with an Intel C++ Compiler Classic (icc/icl) compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select intel for **Compiler**, in the user interface, you see only the processors allowed for a Intel C++ Compiler Classic (icc/icl) compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the intel compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

## Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

## Command-Line Information

**Parameter:** -compiler intel -target
**Value:** x86_64
**Default:** x86_64
**Example (Bug Finder):** polyspace-bug-finder -compiler intel -target x86_64
**Example (Code Prover):** polyspace-code-prover -compiler intel -target x86_64
**Example (Bug Finder Server):** polyspace-bug-finder-server -compiler intel -target x86_64
**Example (Code Prover Server):** polyspace-code-prover-server -compiler intel -target x86_64

# Version History

**Introduced in R2022b**

## See Also

Compiler (-compiler) | Target processor type (-target)

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# MPLAB XC8 C Compiler (-compiler microchip)

Specify MPLAB XC8 C compiler

## Description

Specify microchip for the Compiler (-compiler) option if you compile your code with a MPLAB XC8 C compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select microchip for **Compiler**, in the user interface, you see only the processors allowed for a MPLAB XC8 C compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the microchip compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

## Settings

To see the default sizes in bits for the fundamental types that the target uses, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option -dump-preprocessing-info.

- To override the macro definition, use the option Preprocessor definitions (-D).
- To undefine a macro, use the option Disabled preprocessor definitions (-U).

## Tip

- Polyspace does not support the Atmel families of processors, such as AVR, TinyAVR, MegaAVR, XMEGA, and SAM32.
- Polyspace does not support the CPP/P1 or C18 Microchip front-end. This front-end is activated by the compiler when you compile your code with the C90 version of the Standard.
- If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

## Command-Line Information

**Parameter:** -compiler microchip -target
**Value:** pic
**Default:** pic
**Example (Bug Finder):** polyspace-bug-finder -compiler microchip -target pic
**Example (Code Prover):** polyspace-code-prover -compiler microchip -target pic
**Example (Bug Finder Server):** polyspace-bug-finder-server -compiler microchip -target pic

**Example (Code Prover Server):** `polyspace-code-prover-server -compiler microchip -target pic`

## Version History

**Introduced in R2020a**

### See Also

`Compiler (-compiler)` | `Target processor type (-target)`

**Topics**

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# Renesas Compiler (-compiler renesas)

Specify Renesas compiler

## Description

Specify renesas for the Compiler (-compiler) option if you compile your code with a Renesas compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select renesas for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a Renesas compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine, and certain keyword definitions.

If you specify the renesas compiler, you must specify the path to your compiler header files. See "Provide Standard Library Headers for Polyspace Analysis".

## Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option -dump-preprocessing-info.

- To override the macro definition, use the option Preprocessor definitions (-D).
- To undefine a macro, use the option Disabled preprocessor definitions (-U).

## Command-Line Information

**Parameter:** -compiler renesas -target
**Value:** rl78 | rh850 | rx | sh
**Default:** rl78
**Example (Bug Finder):** polyspace-bug-finder -compiler renesas -target rx
**Example (Code Prover):** polyspace-code-prover -compiler renesas -target rx
**Example (Bug Finder Server):** polyspace-bug-finder-server -compiler renesas -target rx
**Example (Code Prover Server):** polyspace-code-prover-server -compiler renesas -target rx

## Tips

If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

# Version History

**Introduced in R2018b**

## See Also

`Compiler (-compiler)` | `Target processor type (-target)`

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# TASKING Compiler (-compiler tasking)

Specify the Altium TASKING compiler

## Description

Specify `tasking` for `Compiler (-compiler)` if you compile your code using the Altium® TASKING compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `tasking` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for the TASKING compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `tasking` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

The software supports different versions of the TASKING compiler, depending on the target:

- TriCore: 6.x and older versions
- C166: 4.x and older versions
- ARM: 5.x and older versions
- RH850: 2.x and older versions

## Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

## Tips

- Polyspace does not support some constructs specific to the TASKING compiler.

  For the list of unsupported constructs, see `codeprover_limitations.pdf` in *`polyspaceroot`*`\polyspace\verifier\code_prover_desktop`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.
- The CPU used is TC1793. If you use a different CPU, set the following analysis options in your project:

  - `Disabled preprocessor definitions (-U)`: Undefine the macro `__CPU_TC1793B__`.

- `Preprocessor definitions (-D)`: Define the macro `__CPU__`. Enter `__CPU__=`*`xxx`*, where *`xxx`* is the name of your CPU.

  Additionally, define the equivalent of the macro `__CPU_TC1793B__` for your CPU. For instance, enter `__CPU_TC1793A__`.

  Instead of manually specifying your compiler, if you trace your build command (makefile), Polyspace can detect your CPU and add the required definitions in your project.
- For some errors related to TASKING compiler-specific constructs, see solutions in “Fix Polyspace Compilation Errors Related to TASKING Compiler”.
- If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

## Command-Line Information

**Parameter:** `-compiler tasking -target`
**Value:** `tricore | c166 | rh850 | arm`
**Default:** `tricore`
**Example (Bug Finder):** `polyspace-bug-finder -compiler tasking -target tricore`
**Example (Code Prover):** `polyspace-code-prover -compiler tasking -target tricore`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -compiler tasking -target tricore`
**Example (Code Prover Server):** `polyspace-code-prover-server -compiler tasking -target tricore`

# Version History

**Introduced in R2017a**

## See Also

`Compiler (-compiler)` | `Target processor type (-target)`

### Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”

# Texas Instruments Compiler (-compiler ti)

Specify Texas Instruments compiler

## Description

Specify `ti` for `Compiler (-compiler)` if you compile your code using a Texas Instruments compiler. By specifying your compiler, you can avoid compilation errors from syntax that is not part of the Standard but comes from language extensions.

Then, specify your target processor type. If you select `ti` for **Compiler**, in the user interface of the Polyspace desktop products, you see only the processors allowed for a Texas Instruments compiler. Your choice of target processor determines the size of fundamental data types, the endianness of the target machine and certain keyword definitions.

If you specify the `ti` compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis”.

## Settings

To see the default sizes in bits for the fundamental types that the targets use, see the online documentation.

Your compiler specification also determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override the macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine a macro, use the option `Disabled preprocessor definitions (-U)`.

## Tips

Polyspace does not support some constructs specific to the Texas Instruments compiler.

For the list of unsupported constructs, see `codeprover_limitations.pdf` in *`polyspaceroot`*`\polyspace\verifier\code_prover_desktop`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

## Command-Line Information

**Parameter:** `-compiler ti -target`
**Value:** `c28x | c6000 | arm | msp430`
**Default:** `c28x`
**Example (Bug Finder):** `polyspace-bug-finder -compiler ti -target msp430`
**Example (Code Prover):** `polyspace-code-prover -compiler ti -target msp430`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -compiler ti -target msp430`

**Example (Code Prover Server):** `polyspace-code-prover-server -compiler ti -target msp430`

## Version History

**Introduced in R2018a**

## See Also

`Compiler (-compiler)` | `Target processor type (-target)`

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"
"Fix Polyspace Compilation Errors Related to Texas Instruments Compilers"

# Generic target options

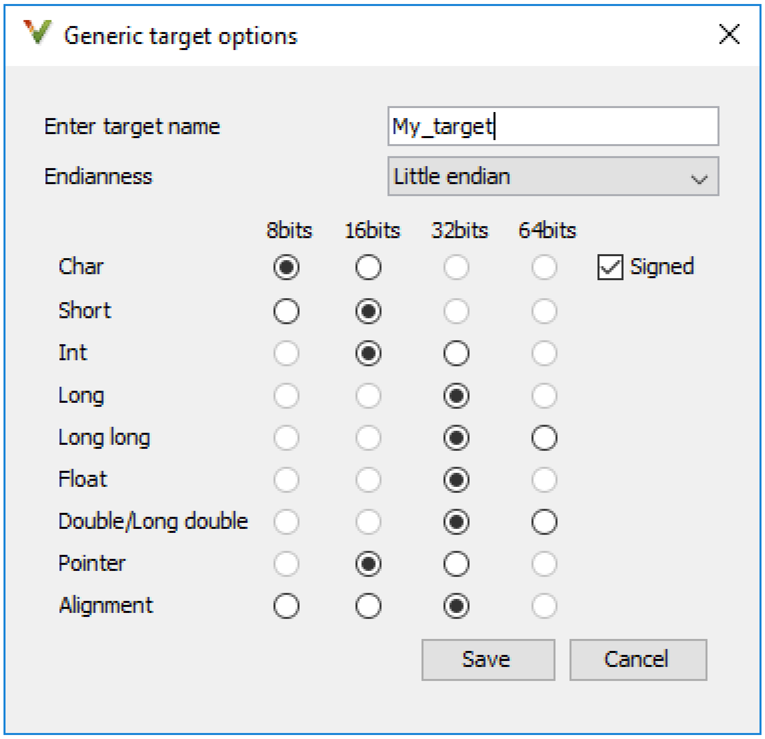
Specify size of data types and endianness by creating your own target processor

## Description

If a target processor is not directly supported by Polyspace, you can create your own target. You specify the target mcpu representing a generic "Micro Controller/Processor Unit" and then explicitly specify sizes of fundamental data types, endianness and other characteristics.

## Settings

In the user interface of the Polyspace desktop products, the **Generic target options** dialog box opens when you set the **Target processor type** to mcpu. The **Target processor type** option is available on the **Target & Compiler** node in the **Configuration** pane.



Generic target options

Enter target name: My_target

Endianness: Little endian

| | 8bits | 16bits | 32bits | 64bits | |
|---|---|---|---|---|---|
| Char | ● | ○ | ○ | ○ | ☑ Signed |
| Short | ○ | ● | ○ | ○ | |
| Int | ○ | ● | ○ | ○ | |
| Long | ○ | ○ | ● | ○ | |
| Long long | ○ | ○ | ● | ○ | |
| Float | ○ | ○ | ● | ○ | |
| Double/Long double | ○ | ○ | ● | ○ | |
| Pointer | ○ | ● | ○ | ○ | |
| Alignment | ○ | ○ | ● | ○ | |

Use the dialog box to specify the name of a new mcpu target, for example My_target. That new target is added to the **Target processor type** option list.

**Default characteristics of a new target:** listed as *type* [size]

- *char* [8]
- *short* [16]
- *int* [16]

- *long* `[32]`
- *long long* `[32]`
- *float* `[32]`
- *double* `[32]`
- *long double* `[32]`
- *pointer* `[16]`
- *alignment* `[32]`
- *char* is signed
- *endianness* is little-endian

## Dependency

A custom target can only be created when `Target processor type (-target)` is set to `mcpu`.

A custom target is not available when `Compiler (-compiler)` is set to one of the `visual*` options.

## Command-Line Options

When using the command line, use `-target mcpu` along with these target specification options.

| Option | Description | Available With | Example |
|---|---|---|---|
| `-little-endian` | Little-endian architectures are Less Significant byte First (LSF). For example: i386.<br>Specifies that the less significant byte of a short integer (e.g. 0x00FF) is stored at the first byte (0xFF) and the most significant byte (0x00) at the second byte. | All targets | `polyspace-bug-finder -target mcpu -little-endian` |

| Option | Description | Available With | Example |
|---|---|---|---|
| `-big-endian` | Big-endian architectures are Most Significant byte First (MSF). For example: SPARC, m68k.<br>Specifies that the most significant byte of a short integer (e.g. 0x00FF) is stored at the first byte (0x00) and the less significant byte (0xFF) at the second byte. | All targets | `polyspace-bug-finder -target mcpu -big-endian` |
| `-default-sign-of-char [signed \| unsigned]` | Specify default sign of `char`.<br>`signed`: Specifies that `char` is signed, overriding target's default.<br>`unsigned`: Specifies that `char` is unsigned, overriding target's default. | All targets | `polyspace-bug-finder -default-sign-of-char unsigned -target mcpu` |
| `-char-is-16bits` | `char` defined as 16 bits and all objects have a minimum alignment of 16 bits<br>Incompatible with `-short-is-8bits` and `-align 8` | `mcpu` | `polyspace-bug-finder -target mcpu -char-is-16bits` |
| `-short-is-8bits` | Define `short` as 8 bits, regardless of sign | `mcpu` | `polyspace-bug-finder -target mcpu -short-is-8bits` |
| `-int-is-32bits` | Define `int` as 32 bits, regardless of sign. Alignment is also set to 32 bits. | `mcpu`, `hc08`, `hc12`, `mpc5xx` | `polyspace-bug-finder -target mcpu -int-is-32bits` |
| `-long-is-32bits` | Define `long` as 32 bits, regardless of sign. Alignment is also set to 32 bits.<br>If your project sets `int` to 64 bits, you cannot use this option. | All targets | `polyspace-bug-finder -target mcpu -long-is-32bits` |

| Option | Description | Available With | Example |
|---|---|---|---|
| `-long-long-is-64bits` | Define `long long` as 64 bits, regardless of sign. Alignment is also set to 64 bits. | `mcpu` | `polyspace-bug-finder -target mcpu -long-long-is-64bits` |
| `-double-is-64bits` | Define `double` and `long double` as 64 bits, regardless of sign. | `mcpu`, `sharc21x61`, `hc08`, `hc12`, `mpc5xx` | `polyspace-bug-finder -target mcpu -double-is-64bits` |
| `-pointer-is-24bits` | Define pointer as 24 bits, regardless of sign. | `c18` | `polyspace-bug-finder -target c18 -pointer-is-24bits` |
| `-pointer-is-32bits` | Define pointer as 32 bits, regardless of sign. | `mcpu` | `polyspace-bug-finder -target mcpu -pointer-is-32bits` |
| `-align [128\|64\|32\|16\|8]` | Specifies the largest alignment of struct or array objects to the 128, 64, 32, 16, or 8 bit boundaries.<br>For instance, if the alignment of basic types in an array or struct is always 8, the array or struct storage is strictly determined by the size of the individual data objects without member and end padding. | All targets | `polyspace-bug-finder -target mcpu -align 16` |

See also:

- `Management of wchar_t (-wchar-t-type-is)`
- `Management of size_t (-size-t-type-is)`
- `Enum type definition (-enum-type-definition)`

You can also use the option `-custom-target` to specify sizes in bytes of fundamental data types, signedness of plain `char`, alignment of structures and underlying types of standard `typedef`-s such as `size_t`, `wchar_t` and `ptrdiff_t`.

## Examples

### GCC Toolchains

If you use any of these GCC toolchains for your software development, you can setup your Polyspace analysis so that your code will compile with Polyspace:

- ARM Ltd's GNU Arm Embedded Toolchain

- HighTec EDV-Systeme
- Linaro® GNU cross-toolchain
- Melexis®
- MENTOR® Embedded Sourcery CodeBench
- QNX® Software Development Platform
- Rowley Associates' CrossWorks
- STMicroelectronics® TrueSTUDIO® for STM32
- Texas Instruments Code Composer Studio™
- Wind River GNU Compiler

Use `polyspace-configure` on a build command that uses one of these toolchains and extract information about your compiler configuration. The command creates a Polyspace project by default. To generate an options file that you then pass to Polyspace at the command line, run `polyspace-configure` with the option `-output-options-file`.

Alternatively, if you prefer to set the details of your compiler configuration manually:

- Select the `gnu#.x` compiler that corresponds to your compiler version for `Compiler (-compiler)`.
- Specify your target by using the “Command-Line Options” on page 2-46. For an example of targets you can specify, see “Targets for GCC Based Compilers” on page 2-49.
- Specify your compiler macro definitions with `Preprocessor definitions (-D)`.

### Targets for GCC Based Compilers

If you select one of the `gnu#.x` compilers for `Compiler (-compiler)`, you can specify one of the supported target processor types. See `Target processor type (-target)`. If a target processor type is not directly listed as supported, you can create the target by using this option.

For instance, you can create these targets:

- **Tricore**: Use these options:

  ```
  -target mcpu
  -int-is-32bits
  -long-long-is-64bits
  -double-is-64bits
  -pointer-is-32bits
  -enum-type-definition auto-signed-first
  -wchar-t-type-is signed-int
  ```
- **PowerPC**: Use these options:

  ```
  -target mcpu
  -int-is-32bits
  -long-long-is-64bits
  -double-is-64bits
  -pointer-is-32bits
  -wchar-t-type-is signed-int
  ```
- **ARM**: Use these options:

  ```
  -target mcpu
  -int-is-32bits
  ```

```
-long-long-is-64bits
-double-is-64bits
-pointer-is-32bits
-enum-type-definition auto-signed-first
-wchar-t-type-is unsigned-int
```

- **MSP430**: Use these options:

```
-target mcpu
-long-long-is-64bits
-double-is-64bits
-wchar-t-type-is signed-long
-align 16
```

### Emulate Microchip MPLAB XC16 and XC32 Compilers

If you build your source code using Microchip MPLAB XC16 or XC32 compilers, you can set up your Polyspace analysis so that your code will compile with Polyspace. Enter these options at the command line or specify them in the **Configuration** pane of the Polyspace desktop user interface.

| Compile r | Target Processor Families | Options |
|---|---|---|
| MPLAB XC16 | PIC24<br>dsPIC | -compiler gnu4.6<br>-D__XC__<br>-D__XC16__<br>-target=mcpu<br>-wchar-t-type-is unsigned-int<br>-align 16<br>-long-long-is-64bits |
| MPLAB XC32 | PIC32 | -compiler gnu4.8<br>-custom-target true,8,2,4,-1,4,8,4,4,8,4,8,1,<br>big,unsigned_long,long,int<br>-D__PIC32M<br>-D__PIC32MX<br>-D__PIC32MX__<br>-D__XC32<br>-D__XC32__<br>-D__XC<br>-D__XC__<br>-D__mips=32<br>-D__mips__<br>-D_mips |

The set of macros specified with the option `Preprocessor definitions (-D)` is a minimal set. Specify additional macros as needed to ensure your code compiles with Polyspace.

## Tips

If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

## See Also

`Target processor type (-target)`

**Topics**

"Specify Polyspace Analysis Options"

"Specify Target Environment and Compiler Behavior"

# Sfr type support (`-sfr-types`)

Specify sizes of `sfr` types for code developed with Keil or IAR compilers

## Description

Specify sizes of `sfr` types (types that define special function registers).

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependency” on page 2-52 for other options you must also enable.

**Command line and options file**: Use the option `-sfr-types`. See “Command-Line Information” on page 2-52.

### Why Use This Option

Use this option if you have statements such as `sfr addr = 0x80;` in your code. `sfr` types are not standard C types. Therefore, you must specify their sizes explicitly for the Polyspace analysis.

## Settings

**No Default**

List each sfr name and its size in bits.

## Dependency

This option is available only when `Compiler (-compiler)` is set to `keil` or `iar`.

## Command-Line Information

**Syntax:** `-sfr-types` *`sfr_name=size_in_bits`*`,...`
**No Default**
**Name Value:** an sfr name such as `sfr16`.
**Size Value:** `8 | 16 | 32`
**Example (Bug Finder):** `polyspace-bug-finder -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...`
**Example (Code Prover):** `polyspace-code-prover -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...`
**Example (Code Prover Server):** `polyspace-code-prover-server -lang c -compiler iar -sfr-types sfr=8,sfr16=16 ...`

## See Also

### Topics
“Specify Polyspace Analysis Options”

“Specify Target Environment and Compiler Behavior”
“Supported Keil or IAR Language Extensions”

# Division round down (`-div-round-down`)

Round down quotients from division or modulus of negative numbers instead of rounding up

## Description

Specify whether quotients from division and modulus of negative numbers are rounded up or down.

**Note** `a = (a / b) * b + a % b` is always true.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

**Command line and options file**: Use the option `-div-round-down`. See “Command-Line Information” on page 2-55.

### Why Use This Option

Use this option to emulate your compiler.

The option is relevant only for compilers following C90 standard (ISO/IEC 9899:1990). The standard stipulates that "*if either operand of / or % is negative, whether the result of the / operator, is the largest integer less or equal than the algebraic quotient or the smallest integer greater or equal than the quotient, is implementation defined, same for the sign of the % operator*". The standard allows compilers to choose their own implementation.

For compilers following the C99 standard ((ISO/IEC 9899:1999), this option is not required. The standard enforces division with rounding towards zero (section 6.5.5).

## Settings

☑ On

If either operand of `/` or `%` is negative, the result of the / operator is the largest integer less than or equal to the algebraic quotient. The result of the `%` operator is deduced from `a % b = a - (a / b) * b`.

*Example*: `assert(-5/3 == -2 && -5%3 == 1);` is true.

□ Off (default)

If either operand of `/` or `%` is negative, the result of the `/` operator is the smallest integer greater than or equal to the algebraic quotient. The result of the `%` operator is deduced from `a % b = a - (a / b) * b`.

This behavior is also known as rounding towards zero.

*Example*: `assert(-5/3 == -1 && -5%3 == -2);` is true.

## Command-Line Information

**Parameter:** `-div-round-down`
**Default:** Off
**Example (Bug Finder):** `polyspace-bug-finder -div-round-down`
**Example (Code Prover):** `polyspace-code-prover -div-round-down`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -div-round-down`
**Example (Code Prover Server):** `polyspace-code-prover-server -div-round-down`

## See Also

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# Enum type definition (-enum-type-definition)

Specify how to represent an enum with a base type

## Description

Allow the analysis to use different base types to represent an enumerated type, depending on the enumerator values and the selected definition. When using this option, each enum type is represented by the smallest integral type that can hold its enumeration values.

This option is available on the **Target & Compiler** node in the **Configuration** pane.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

**Command line and options file**: Use the option -enum-type-definition. See “Command-Line Information” on page 2-57.

### Why Use This Option

Your compiler represents enum variables as constants of a base integer type. Use this option so that you can emulate your compiler.

To check your compiler settings:

1 Compile this code using the compiler settings that you typically use:

```
enum { MAXSIGNEDBYTE=127 } mysmallenum_t;

int dummy[(int)sizeof(mysmallenum_t) - (int)sizeof(int)];
```

   If compilation fails, you have to use one of auto-signed-first or auto-unsigned-first.

2 Compile this code using the compiler settings that you typically use:

```
#include <limits.h>

enum { MYINTMAX = INT_MAX } myintenum_t;

int dummy[(MYINTMAX + 1) < 0 ? -1:1];
```

   If compilation fails, use auto-signed-first for this option, otherwise use auto-unsigned-first.

## Settings

**Default:** defined-by-compiler

defined-by-compiler

   Uses the signed integer type for all compilers except gnu, clang and tasking.

For the gnu and clang compilers, it uses the first type that can hold all of the enumerator values from this list: `unsigned int`, `signed int`, `unsigned long`, `signed long`, `unsigned long long` and `signed long long`.

For the tasking compiler, it uses the first type that can hold all of the enumerator values from this list: `char`, `unsigned char`, `short`, `unsigned short`, `int`, and `unsigned int`.

`auto-signed-first`

Uses the first type that can hold all of the enumerator values from this list: `signed char`, `unsigned char`, `signed short`, `unsigned short`, `signed int`, `unsigned int`, `signed long`, `unsigned long`, `signed long long`, and `unsigned long long`.

`auto-unsigned-first`

Uses the first type that can hold all of the enumerator values from these lists:

- If enumerator values are positive: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`.
- If one or more enumerator values are negative: `signed char`, `signed short`, `signed int`, `signed long`, and `signed long long`.

## Command-Line Information

**Parameter:** `-enum-type-definition`
**Value:** `defined-by-compiler | auto-signed-first | auto-unsigned-first`
**Default:** `defined-by-compiler`
**Example (Bug Finder):** `polyspace-bug-finder -enum-type-definition auto-signed-first`
**Example (Code Prover):** `polyspace-code-prover -enum-type-definition auto-signed-first`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -enum-type-definition auto-signed-first`
**Example (Code Prover Server):** `polyspace-code-prover-server -enum-type-definition auto-signed-first`

## See Also

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# Signed right shift (-logical-signed-right-shift)

Specify how to treat the sign bit for logical right shifts on signed variables

## Description

Choose between arithmetic and logical shift for right shift operations on negative values.

This option does not modify compile-time expressions. For more details, see "Limitation" on page 2-58.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

**Command line and options file**: Use the option `-logical-signed-right-shift`. See "Command-Line Information" on page 2-59.

### Why Use This Option

The C99 Standard (sec 6.5.7) states that for a right-shift operation `x1>>x2`, if `x1` is signed and has negative values, the behavior is implementation-defined. Different compilers choose between arithmetic and logical shift. Use this option to emulate your compiler.

## Settings

**Default:** `Arithmetical`

`Arithmetical`

The sign bit remains:

```
(-4) >> 1 = -2
(-7) >> 1 = -4
  7  >> 1 = 3
```

`Logical`

0 replaces the sign bit:

```
(-4) >> 1 = (-4U) >> 1 = 2147483646
(-7) >> 1 = (-7U) >> 1 = 2147483644
  7  >> 1 = 3
```

## Limitation

In compile-time expressions, this Polyspace option does not change the standard behavior for right shifts.

For example, consider this right shift expression:

```
int arr[ ((-4) >> 20) ];
```

The compiler computes array sizes, so the expression `(-4) >> 20` is evaluated at compilation time. Logically, this expression is equivalent to 4095. However, arithmetically, the result is -1. This statement causes a compilation error (arrays cannot have negative size) because the standard right-shift behavior for signed integers is arithmetic.

## Command-Line Information

When using the command line, arithmetic is the default computation mode. When this option is set, logical computation is performed.
**Parameter:** `-logical-signed-right-shift`
**Default:** Arithmetic signed right shifts
**Example (Bug Finder):** `polyspace-bug-finder -logical-signed-right-shift`
**Example (Code Prover):** `polyspace-code-prover -logical-signed-right-shift`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -logical-signed-right-shift`
**Example (Code Prover Server):** `polyspace-code-prover-server -logical-signed-right-shift`

## See Also

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# Block char16/32_t types (-no-uliterals)

Disable Polyspace definitions for char16_t or char32_t

## Description

Specify that the analysis must not define char16_t or char32_t types.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See “Dependencies” on page 2-60 for other options you must also enable.

**Command line and options file**: Use the option -no-uliterals. See “Command-Line Information” on page 2-61.

### Why Use This Option

If your compiler defines char16_t and/or char32_t through a typedef statement or by using includes, use this option to turn off the standard Polyspace definition of char16_t and char32_t.

To check if your compiler defines these types, save the following code in a C++ file and compile this code using the compiler settings that you typically use:

```
typedef unsigned short char16_t;
typedef unsigned long char32_t;
```

See if there is a compilation error.

- If the file fails to compile with an error such as redeclaration of C++ built-in type, it means that your compiler has the types char16_t and char32_t defined as built-in types. Enable this Polyspace option to emulate your compiler behavior.
- If the file compiles, it means that your compiler defines these types through typedef-s in includes. If you have already added those includes to your project, you do not need to enable this Polyspace option.

## Settings

☑ On

The analysis does not allow char16_t and char32_t types.

□ Off (default)

The analysis allows char16_t and char32_t types.

## Dependencies

You can select this option only when these conditions are true:

- Source code language (-lang) is set to CPP or C-CPP.

- `Compiler (-compiler)` is set to `generic` or a `gnu` version.

## Command-Line Information

**Parameter:** `-no-uliterals`
*Default:* off
**Example (Bug Finder):** `polyspace-bug-finder -lang cpp -compiler gnu4.7 -cpp-version cpp11 -no-uliterals`
**Example (Code Prover):** `polyspace-code-prover -compiler gnu4.7 -lang cpp -cpp-version cpp11 -no-uliterals`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -lang cpp -compiler gnu4.7 -cpp-version cpp11 -no-uliterals`
**Example (Code Prover Server):** `polyspace-code-prover-server -compiler gnu4.7 -lang cpp -cpp-version cpp11 -no-uliterals`

## See Also

`Compiler (-compiler)`

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# Pack alignment value (`-pack-alignment-value`)

Specify default structure packing alignment for code developed in Visual C++

## Description

Specify the default packing alignment (in bytes) for structures, unions, and class members.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

**Command line and options file**: Use the option `-pack-alignment-value`. See “Command-Line Information” on page 2-62.

### Why Use This Option

If you use compiler options to specify how members of a structure are packed into memory, use this option to emulate your compiler.

For instance, if you use the Visual Studio option /Zp to specify an alignment, use this option for your Polyspace analysis.

If you use `#pragma pack` directives in your code to specify alignment, and also specify this option for analysis, the `#pragma pack` directives take precedence.

## Settings

**Default**: 8

You can enter one of these values:

- 1
- 2
- 4
- 8
- 16

## Command-Line Information

**Parameter:** `-pack-alignment-value`
**Value:** `1 | 2 | 4 | 8 | 16`
**Default:** 8
**Example (Bug Finder):** `polyspace-bug-finder -compiler visual10 -pack-alignment-value 4`
**Example (Code Prover):** `polyspace-code-prover -compiler visual10 -pack-alignment-value 4`

**Example (Bug Finder Server):** `polyspace-bug-finder-server -compiler visual10 -pack-alignment-value 4`
**Example (Code Prover Server):** `polyspace-code-prover-server -compiler visual10 -pack-alignment-value 4`

## See Also

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"
"Code Prover Assumptions About #pragma Directives" (Polyspace Code Prover)

# Ignore pragma pack directives (-ignore-pragma-pack)

Ignore #pragma pack directives

## Description

Specify that the analysis must ignore #pragma pack directives in the code.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

**Command line and options file**: Use the option -ignore-pragma-pack. See “Command-Line Information” on page 2-64.

### Why Use This Option

Use this option if #pragma pack directives in your code cause linking errors.

For instance, you have two structures with the same name in your code, but one declaration follows a #pragma pack(2) statement. Because the default alignment is 8 bytes, the different packing for the two structures causes a linking error. Use this option to avoid such errors.

## Settings

☑ On

The analysis ignores the #pragma directives.

☐ Off (default)

The analysis takes into account specifications in the #pragma directives.

## Command-Line Information

**Parameter:** -ignore-pragma-pack
**Default**: Off
**Example (Bug Finder):** polyspace-bug-finder -ignore-pragma-pack
**Example (Code Prover):** polyspace-code-prover -ignore-pragma-pack
**Example (Bug Finder Server):** polyspace-bug-finder-server -ignore-pragma-pack
**Example (Code Prover Server):** polyspace-code-prover-server -ignore-pragma-pack

## See Also

### Topics

“Specify Polyspace Analysis Options”
“Specify Target Environment and Compiler Behavior”
“Code Prover Assumptions About #pragma Directives” (Polyspace Code Prover)

# Management of size_t (-size-t-type-is)

Specify the underlying data type of size_t

## Description

Specify the underlying data type of size_t explicitly: unsigned char, unsigned short, unsigned int, unsigned long or unsigned long long. If you do not specify this option, your choice of compiler determines the underlying type.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

**Command line and options file**: Use the option -size-t-type-is. See “Command-Line Information” on page 2-66.

### Why Use This Option

The analysis associates a data type with size_t when you specify your compiler using the option Compiler (-compiler). In most cases, you do not have to explicitly use this option and specify an underlying type for size_t.

In some situations, when building your code, you might be using a compiler option that changed the compiler's default definition of size_t. In these cases, emulate your compiler option by using this Polyspace analysis option. Otherwise, you might see an error message related to size_t during Polyspace analysis. If you see such an error message, to probe further and determine the underlying type of size_t, compile this code with your compiler using the options that you typically use:

```
/* Header defines malloc as void* malloc (size_t size)
#include <stlib.h>

void* malloc (unsigned int size);
```

If the file does not compile, your compiler (along with compiler options) defines size_t using a type different from unsigned int. Replace unsigned int with another type such as unsigned long and try again till you determine the underlying type of size_t.

## Settings

**Default:** defined-by-compiler

defined-by-compiler
: Your specification for Compiler (-compiler) determines the underlying type of size_t.

unsigned-int
: The analysis considers unsigned int as the underlying type of size_t.

unsigned-long
: The analysis considers unsigned long as the underlying type of size_t.

`unsigned-long-long`

The analysis considers `unsigned long long` as the underlying type of `size_t`.

`unsigned-char`

The analysis considers `unsigned char` as the underlying type of `size_t`.

`unsigned-short`

The analysis considers `unsigned short` as the underlying type of `size_t`.

## Tips

Compilation errors from incorrect definition of `size_t` can appear in unexpected ways. For instance, you might see an error like this:

```
first parameter of allocation function must be of type "size_t"
```

on a declaration of an allocation function such as:

```
void * operator new(size_t size);
```

This error appears because Polyspace internally declares the allocation function with the `size_t` definition from your Polyspace analysis configuration, but your declaration might be using a different `size_t` definition from a compiler header. The mismatch in the `size_t` definitions leads to a mismatch in the declarations of the allocation functions and shows up as an error message about the allocation functions.

## Command-Line Information

**Parameter:** `-size-t-type-is`
**Value:** `defined-by-compiler` | `unsigned-char` | `unsigned-int` | `unsigned-short` | `unsigned-long` | `unsigned-long-long`
**Default:** `defined-by-compiler`
**Example (Bug Finder):** `polyspace-bug-finder -size-t-type-is unsigned-long`
**Example (Code Prover):** `polyspace-code-prover -size-t-type-is unsigned-long`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -size-t-type-is unsigned-long`
**Example (Code Prover Server):** `polyspace-code-prover-server -size-t-type-is unsigned-long`

## See Also

`-custom-target`

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# Management of wchar_t (-wchar-t-type-is)

Specify the underlying data type of `wchar_t`

## Description

Specify the underlying data type of `wchar_t` explicitly. If you do not specify this option, your choice of compiler determines the underlying type.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Target & Compiler** node.

**Command line and options file**: Use the option `-wchar-t-type-is`. See “Command-Line Information” on page 2-67.

### Why Use This Option

The analysis associates a data type with `wchar_t` when you specify your compiler. If you use a compiler option that changes this default type, emulate your compiler option by using this analysis option.

## Settings

**Default:** `defined-by-compiler`

`defined-by-compiler`

Your specification for `Compiler (-compiler)` determines the underlying type of `wchar_t`.

`signed-short`

The analysis considers `signed short` as the underlying type of `wchar_t`.

`unsigned-short`

The analysis considers `unsigned short` as the underlying type of `wchar_t`.

`signed-int`

The analysis considers `signed int` as the underlying type of `wchar_t`.

`unsigned-int`

The analysis considers `unsigned int` as the underlying type of `wchar_t`.

`signed-long`

The analysis considers `signed long` as the underlying type of `wchar_t`.

`unsigned-long`

The analysis considers `unsigned long` as the underlying type of `wchar_t`.

## Command-Line Information

**Parameter:** `-wchar-t-type-is`

**Value:** `defined-by-compiler | signed-short | unsigned-short | signed-int | unsigned-int | signed-long | unsigned-long`
**Default:** `defined-by-compiler`
**Example (Bug Finder):** `polyspace-bug-finder -wchar-t-type-is signed-int`
**Example (Code Prover):** `polyspace-code-prover -wchar-t-type-is signed-int`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -wchar-t-type-is signed-int`
**Example (Code Prover Server):** `polyspace-code-prover-server -wchar-t-type-is signed-int`

## See Also

### Topics

"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"

# Ignore link errors (-no-extern-C)

Ignore certain linking errors

## Description

Specify that the analysis must ignore certain linking errors.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Environment Settings** node. See "Dependency" on page 2-69 for other options that you must also enable.

**Command line and options file**: Use the option `-no-extern-C`. See "Command-Line Information" on page 2-69.

### Why Use This Option

Some functions may be declared inside an `extern "C" { }` block in some files and not in others. Then, their linkage is not the same and it causes a link error according to the ANSI standard.

Applying this option will cause Polyspace to ignore this error. This permissive option may not resolve all the extern C linkage errors.

## Settings

☑ On

Ignore linking errors if possible.

□ Off (default)

Stop analysis for linkage errors.

## Dependency

This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.

## Command-Line Information

**Parameter:** `-no-extern-C`
**Default:** off
**Example (Bug Finder):** `polyspace-bug-finder -lang cpp -no-extern-C`
**Example (Code Prover):** `polyspace-code-prover -lang cpp -no-extern-C`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -lang cpp -no-extern-C`
**Example (Code Prover Server):** `polyspace-code-prover-server -lang cpp -no-extern-C`

## See Also

### Topics

"Specify Polyspace Analysis Options"

# Preprocessor definitions (-D)

Replace macros in preprocessed code

## Description

Replace macros with their definitions in preprocessed code.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Macros** node.

**Command line and options file**: Use the option `-D`. See “Command-Line Information” on page 2-72.

### Why Use This Option

Use this option to emulate your compiler behavior. For instance, if your compiler considers a macro `_WIN32` as defined when you build your code, it executes code in a `#ifdef _WIN32` statement. If Polyspace does not consider that macro as defined, you must use this option to replace the macro with 1.

Depending on your settings for `Compiler (-compiler)`, some macros are defined by default. Use this option to define macros that are not implicitly defined.

Typically, you recognize from compilation errors that a certain macro is not defined. For instance, the following code does not compile if the macro `_WIN32` is not defined.

```
#ifdef _WIN32
  int env_var;
#endif

void set() {
  env_var=1;
}
```

The error message states that `env_var` is undefined. However, the definition of `env_var` is in the `#ifdef _WIN32` statement. The underlying cause for the error is that the macro `_WIN32` is not defined. You must define `_WIN32`.

## Settings

### No Default

Using the ✚ button, add a row for the macro you want to define. The definition must be in the format *`Macro`*`=`*`Value`*. If you want Polyspace to ignore the macro, leave the *`Value`* blank.

For example:

- name1=name2 replaces all instances of name1 by name2.
- name= instructs the software to ignore name.
- name with no equals sign or value replaces all instances of name by 1. To define a macro to execute code in a #ifdef *macro_name* statement, use this syntax.

## Tips

- If Polyspace does not support a non-ANSI keyword and shows a compilation error, use this option to replace all occurrences of the keyword with a blank string in preprocessed code. The replacement occurs only for the purposes of the analysis. Your original source code remains intact.

  For instance, if your compiler supports the __far keyword, to avoid compilation errors:

  - In the user interface (desktop products only), enter __far=.
  - On the command line, use the flag -D __far=.

  The software replaces the __far keyword with a blank string during preprocessing. For example:

  ```
  int __far* pValue;
  ```

  is converted to:

  ```
  int * pValue;
  ```
- Polyspace recognizes keywords such as restrict and does not allow their use as identifiers. If you use those keywords as identifiers (because your compiler does not recognize them as keywords), replace the disallowed name with another name using this option. The replacement occurs only for the purposes of the analysis. Your original source code remains intact.

  For instance, to allow use of restrict as identifier:

  - In the user interface, enter restrict=my_restrict.
  - On the command line, use the flag -D restrict=my_restrict.
- Your compiler specification determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option -dump-preprocessing-info.

  - To override the macro definition coming from a compiler specification, use this option.
  - To undefine a macro, use the option Disabled preprocessor definitions (-U).

## Command-Line Information

You can specify only one flag with each -D option. However, you can specify the option multiple times.
**Parameter:** -D
**No Default**
**Value:** *flag=value*
**Example (Bug Finder):** polyspace-bug-finder -D HAVE_MYLIB -D int32_t=int
**Example (Code Prover):** polyspace-code-prover -D HAVE_MYLIB -D int32_t=int
**Example (Bug Finder Server):** polyspace-bug-finder-server -D HAVE_MYLIB -D int32_t=int
**Example (Code Prover Server):** polyspace-code-prover-server -D HAVE_MYLIB -D int32_t=int

## See Also

`Disabled preprocessor definitions (-U)`

### Topics

"Specify Polyspace Analysis Options"

# Disabled preprocessor definitions (-U)

Undefine macros in preprocessed code

## Description

Undefine macros in preprocessed code.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Macros** node.

**Command line and options file**: Use the option -U. See “Command-Line Information” on page 2-75.

### Why Use This Option

Use this option to emulate your compiler behavior. For instance, your compiler might consider a macro _WIN32 as undefined and execute the code in a #ifndef _WIN32 block. To emulate this behavior when running a Polyspace analysis, use this option to specify _WIN32 as undefined.

This option undefines these macros:

- Macros you defined by using the -D option. See Preprocessor definitions (-D).
- Macros that might be implicitly defined by the compiler.
- Macros that Polyspace enables by default to emulate compiler behavior. See Compiler (-compiler).

If you define a macro by using a #define statement in your source code, this option cannot undefine it.

Typically, you recognize from compilation errors that a certain macro must be undefined. For instance, the following code does not compile if the macro _WIN32 is defined.

```
#ifndef _WIN32
  int env_var;
#endif

void set() {
  env_var=1;
}
```

The error message states that env_var is undefined. However, the definition of env_var is in the #ifndef _WIN32 statement. The underlying cause for the error is that the macro _WIN32 is defined. You must undefine _WIN32.

## Settings

**No Default**

Using the ✚ button, add a new row for each macro being undefined.

## Tips

Your compiler specification determines the values of many compiler-specific macros. In case you want to know how Polyspace defines a specific macro, use the option `-dump-preprocessing-info`.

- To override a macro definition coming from a compiler specification, use the option `Preprocessor definitions (-D)`.
- To undefine the macro, use this option.

## Command-Line Information

You can specify only one flag with each `-U` option. However, you can specify the option multiple times.
**Parameter:** `-U`
**No Default**
**Value:** *`macro`*
**Example (Bug Finder):** `polyspace-bug-finder -U HAVE_MYLIB -U USE_COM1`
**Example (Code Prover):** `polyspace-code-prover -U HAVE_MYLIB -U USE_COM1`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -U HAVE_MYLIB -U USE_COM1`
**Example (Code Prover Server):** `polyspace-code-prover-server -U HAVE_MYLIB -U USE_COM1`

## See Also

`Preprocessor definitions (-D)`

### Topics

"Specify Polyspace Analysis Options"

# Source code encoding (`-sources-encoding`)

Specify the encoding of source files

## Description

Specify the encoding of the source files that you analyze with Polyspace.

Use this option only if you see compilation errors or display issues from non-ASCII characters in your source files. The option forces an internal conversion of your source files from the specified encoding to an UTF-8 encoding and might help resolve the issue.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Environment Settings** node.

**Command line and options file**: Use the option `-sources-encoding`. See “Command-Line Information” on page 2-77.

### Why Use This Option

The analysis uses the default encoding of your operating system as the source code encoding. In most cases, if your source code contains non-ASCII characters, for instance, Japanese or Korean characters, the Polyspace analysis can interpret the characters and later display the source code correctly.

If you still have compilation errors or display issues from non-ASCII characters, you might be using an encoding that is different from the default encoding. You can then specify your source code encoding explicitly using this option.

## Settings

**Default:** `system`

`system`

The analysis uses the default encoding of the operating system.

`shift-jis`

The analysis uses the Shift JIS (Shift Japanese Industrial Standards) encoding, a character encoding for the Japanese language.

`iso-8859-1`

The analysis uses the ISO/IEC 8859-1:1998 encoding, a character encoding that encodes what it refers to as "Latin alphabet no.1", consisting of 191 characters from the Latin script.

`windows-1252`

The analysis uses the Windows-1252 encoding, a single-byte character encoding of the Latin alphabet, used by default in the legacy components of Windows for English and some other Western languages.

`UTF-8`

The analysis uses the UTF-8 encoding, a variable width character encoding capable of encoding all valid code points in Unicode.

Polyspace supports many more encodings. To specify an encoding that is not in the above list in the Polyspace user interface, enter `-sources-encoding` *`encodingname`* in the `Other` field. In particular, if your source files contain a mix of different encodings, you can use `-sources-encoding auto`. In this mode, the analysis uses internal heuristics to determine the encoding of your source files from their contents.

For the full list of supported encodings, at the command line, enter:

```
-list-all-values -sources-encoding
```

with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command. Pipe the output to a file and search the file for the encoding that you are using.

## Command-Line Information

**Parameter:** `-sources-encoding`
**Default:** `system`
**Value:** `auto` | `system` | `shift-jis` | `iso-8859-1` | `windows-1252` | `UTF-8`
**Example (Bug Finder):** `polyspace-bug-finder -sources-encoding windows-1252`
**Example (Code Prover):** `polyspace-code-prover -sources-encoding windows-1252`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources-encoding windows-1252`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources-encoding windows-1252`

Polyspace supports many more encodings besides the above list. For the full list of supported encodings, at the command line, enter:

```
-list-all-values -sources-encoding
```

with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command. Pipe the output to a file and search the file for the encoding that you are using.

## See Also

### Topics

“Specify Polyspace Analysis Options”

# Code from DOS or Windows file system (-dos)

Consider that file paths are in MS-DOS style

## Description

Specify that DOS or Windows files are provided for analysis.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Environment Settings** node.

**Command line and options file**: Use the option `-dos`. See "Command-Line Information" on page 2-78.

### Why Use This Option

Use this option if the contents of the **Include** or **Source** folder come from a DOS or Windows file system. The option helps you resolve case sensitivity and control character issues.

## Settings

☑ On (default)

Analysis understands file names and include paths for Windows/DOS files

For example, with this option,

```
#include "..\mY_TEst.h"^M
```

```
#include "..\mY_other_FILE.H"^M
```

resolves to:

```
#include "../my_test.h"
```

```
#include "../my_other_file.h"
```

In this mode, you see an error if your include folder has header files whose names differ only in case.

□ Off

Characters are not controlled for files names or paths.

## Command-Line Information

**Parameter:** `-dos`
**Default:** Off
**Example (Bug Finder):** `polyspace-bug-finder -dos -I ./my_copied_include_dir -D test=1`

**Example (Code Prover):** `polyspace-code-prover -dos -I ./my_copied_include_dir -D test=1`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -dos -I ./my_copied_include_dir -D test=1`
**Example (Code Prover Server):** `polyspace-code-prover-server -dos -I ./my_copied_include_dir -D test=1`

## See Also

### Topics

"Specify Polyspace Analysis Options"

# Stop analysis if a file does not compile (`-stop-if-compile-error`)

Specify that a compilation error must stop the analysis

## Description

Specify that even a single compilation error must stop the analysis.

### Set Option

**User interface** (desktop products only): In the **Configuration** pane, the option is on the **Environment Settings** node.

**Command line and options file**: Use the option `-stop-if-compile-error`. See “Command-Line Information” on page 2-81.

### Why Use This Option

Use this option to first resolve all compilation errors and then perform the Polyspace analysis. This sequence ensures that all files are analyzed.

Otherwise, only files without compilation errors are fully analyzed. The analysis might return some results for files that do not compile. If a file with compilation errors contains a function definition, the analysis considers the function undefined. This assumption can sometimes make the analysis less precise.

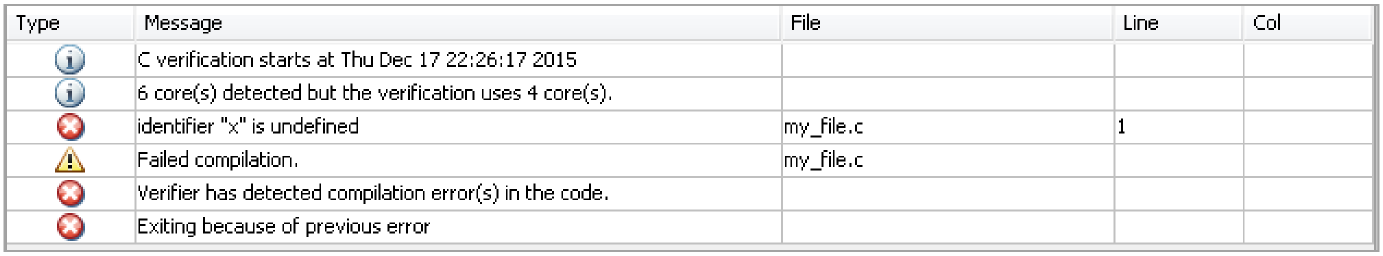
The option is more useful for a Code Prover analysis because the Code Prover run-time checks rely more heavily on range propagation across functions.

## Settings

☑ On

The analysis stops even if a single compilation error occurs.

In the user interface of the Polyspace desktop products, you see the compilation errors on the **Output Summary** pane.



| Type | Message | File | Line | Col |
|---|---|---|---|---|
| ⓘ | C verification starts at Thu Dec 17 22:26:17 2015 | | | |
| ⓘ | 6 core(s) detected but the verification uses 4 core(s). | | | |
| ⊗ | identifier "x" is undefined | my_file.c | 1 | |
| ⚠ | Failed compilation. | my_file.c | | |
| ⊗ | Verifier has detected compilation error(s) in the code. | | | |
| ⊗ | Exiting because of previous error | | | |

For information on how to resolve the errors, see “Troubleshoot Compilation Errors”.

You can also see the errors in the analysis log, a text file generated during the analysis. The log is named `Polyspace_R20##n_ProjectName_date-time.log` and contains lines starting with `Error:` indicating compilation errors. To view the log from the analysis results:

- In the user interface of the Polyspace desktop products, select **Window** > **Show/Hide View** > **Run Log**.
- In the Polyspace Access web interface, open the **Review** tab. Select **Window** > **Run Log**.

Despite compilation errors, you can see some analysis results, for instance, coding rule violations.

□ Off (default)

The analysis does not stop because of compilation errors, but only files without compilation errors are analyzed. The analysis does not consider files that do not compile. If a file with compilation errors contains a function definition, the analysis considers the function undefined. If the analysis needs the definition of such a function, it makes broad assumptions about the function.

- The function return value can take any value in the range allowed by its data type.
- The function can modify arguments passed by reference so that they can take any value in the range allowed by their data types.

If the assumptions are too broad, the analysis can be less precise. For instance, a run-time check can flag an operation in orange even though it does not fail in practice.

If compilation errors occur, in the user interface of the Polyspace desktop products, the **Dashboard** pane has a link, which shows that some files failed to compile. You can click the link and see the compilation errors on the **Output Summary** pane.

You can also see the errors in the analysis log, a text file generated during the analysis. The log is named `Polyspace_R20##n_ProjectName_date-time.log` and contains lines starting with `Error:` indicating compilation errors. To view the log from the analysis results:

- In the user interface of the Polyspace desktop products, select **Window** > **Show/Hide View** > **Run Log**.
- In the Polyspace Access web interface, open the **Review** tab. Select **Window** > **Run Log**.

## Command-Line Information

**Parameter:**`-stop-if-compile-error`
**Default:** Off
**Example (Bug Finder):** `polyspace-bug-finder -sources filename -stop-if-compile-error`
**Example (Code Prover):** `polyspace-code-prover -sources filename -stop-if-compile-error`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources filename -stop-if-compile-error`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources filename -stop-if-compile-error`

# Version History

**Introduced in R2017a**

## See Also

`File does not compile`

### Topics

"Specify Polyspace Analysis Options"

# Command/script to apply to preprocessed files (`-post-preprocessing-command`)

Specify command or script to run on source files after preprocessing phase of analysis

## Description

Specify a command or script to run on each source file after preprocessing.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Environment Settings** node.

**Command line and options file**: Use the option `-post-preprocessing-command`. See "Command-Line Information" on page 2-85.

### Why Use This Option

You can run scripts on preprocessed files to work around compilation errors or imprecisions of the analysis while keeping your original source files untouched. For instance, suppose Polyspace does not recognize a compiler-specific keyword. If you are certain that the keyword is not relevant for the analysis, you can run a Perl script to remove all instances of the keyword. When you use this option, the software removes the keyword from your preprocessed code but keeps your original code untouched.

Use a script only if the existing analysis options do not meet your requirements. For instance:

- For direct replacement of one keyword with another, use the option `Preprocessor definitions (-D)`.

  However, the option does not allow search and replacement involving regular expressions. For regular expressions, use a script.
- For mapping your library function to a standard library function, use the option `-code-behavior-specifications`.

  However, the option supports mapping to only a subset of standard library functions. To map to an unsupported function, use a script.

*If you are unsure about removing or replacing an unsupported construct, do not use this option.* Contact MathWorks® Support for guidance.

## Settings

**No Default**

Enter full path to the command or script or click 📁 to navigate to the location of the command or script. This script is executed before verification.

## Tips

- Your script must be designed to process the standard output from preprocessing and produce its results in accordance with that standard output.
- Your script must preserve the number of lines in the preprocessed file. In other words, it must not add or remove entire lines to or from the file.

  Adding a line or removing one can potentially result in some unpredictable behavior on the location of checks and macros in the Polyspace user interface.
- For a Perl script, in Windows, specify the full path to the Perl executable followed by the full path to the script.

  For example:

  - To specify a Perl command that replaces all instances of the `far` keyword, enter *`polyspaceroot`*`\sys\perl\win32\bin\perl.exe -p -e "s/far//g"`.
  - To specify a Perl script `replace_keyword.pl` that replaces all instances of a keyword, enter *`polyspaceroot`*`\sys\perl\win32\bin\perl.exe` *`absolute_path`* `\replace_keyword.pl`.

  Here, *`polyspaceroot`* is the location of the current Polyspace installation such as `C:\Program Files\Polyspace\R2019a\` and *`absolute_path`* is the location of the Perl script. If the paths contain spaces, use quotes to enclose the full path names.
- Use this Perl script as template. The script removes all instances of the `far` keyword.

```
#!/usr/bin/perl

binmode STDOUT;

# Process every line from STDIN until EOF
while ($line = <STDIN>)
{

  # Remove far keyword
  $line =~ s/far//g;

  # Print the current processed line to STDOUT
  print $line;
}
```

  You can use Perl regular expressions to perform substitutions. For instance, you can use the following expressions.

| Expression | Meaning |
|---|---|
| `.` | Matches any single character except newline |
| `[a-z0-9]` | Matches any single letter in the set `a-z`, or digit in the set `0-9` |
| `[^a-e]` | Matches any single letter not in the set `a-e` |
| `\d` | Matches any single digit |
| `\w` | Matches any single alphanumeric character or _ |
| `x?` | Matches 0 or 1 occurrence of `x` |

| Expression | Meaning |
|---|---|
| x* | Matches 0 or more occurrences of x |
| x+ | Matches 1 or more occurrences of x |

For complete list of regular expressions, see Perl documentation.

## Command-Line Information

**Parameter:** `-post-preprocessing-command`
**Value:** Path to executable file or script in quotes
**No Default**
**Example in Linux® (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-post-preprocessing-command `pwd`/replace_keyword.pl`
**Example in Linux (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-post-preprocessing-command `pwd`/replace_keyword.pl`
**Example in Linux (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-post-preprocessing-command `pwd`/replace_keyword.pl`
**Example in Linux (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-post-preprocessing-command `pwd`/replace_keyword.pl`
**Example in Windows:** `polyspace-bug-finder -sources` *`file_name`* `-post-preprocessing-command "C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl"`

You can specify an absolute path to a file or a path relative to the location from which you run the `polyspace-bug-finder` or `polyspace-code-prover` command. Note that if you are running Perl scripts, in Windows, enter the full path to the Perl executable.

## See Also

`Command/script to apply after the end of the code verification (-post-analysis-command)` | `-regex-replace-rgx -regex-replace-fmt`

### Topics

“Specify Polyspace Analysis Options”
“Remove or Replace Keywords Before Compilation”

# Include (`-include`)

Specify files to be `#include`-ed by each C file in analysis

## Description

Specify files to be `#include`-ed by each C file involved in the analysis. The software enters the `#include` statements in the preprocessed code used for analysis, but does not modify the original source code.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Environment Settings** node.

**Command line and options file**: Use the option `-include`. See “Command-Line Information” on page 2-86.

### Why Use This Option

There can be many reasons why you want to `#include` a file in all your source files.

For instance, you can collect in one header file all workarounds for compilation errors. Use this option to provide the header file for analysis. Suppose you have compilation issues because Polyspace does not recognize certain compiler-specific keywords. To work around the issues, `#define` the keywords in a header file and provide the header file with this option.

## Settings

### No Default

Specify the file name to be included in every file involved in the analysis.

Polyspace still acts on other directives such as `#include <include_file.h>`.

## Command-Line Information

**Parameter:** `-include`
**Default:** None
**Value:** *`file`* (Use `-include` multiple times for multiple files)
**Example (Bug Finder):** `polyspace-bug-finder -include `pwd`/sources/a_file.h -include /inc/inc_file.h`
**Example (Code Prover):** `polyspace-code-prover -include `pwd`/sources/a_file.h -include /inc/inc_file.h`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -include `pwd`/sources/a_file.h -include /inc/inc_file.h`
**Example (Code Prover Server):** `polyspace-code-prover-server -include `pwd`/sources/a_file.h -include /inc/inc_file.h`

## See Also

### Topics

"Specify Polyspace Analysis Options"
"Gather Compilation Options Efficiently"

# Include folders (`-I`)

View include folders used for analysis

## Description

*This option is relevant only for the user interface of the Polyspace desktop products.*

View the include folders used for analysis.

### Set Option

This is not an option that you set in your project configuration. You can only view the include folders in the configuration associated with a result. For instance, in the user interface:

- To add include folders, on the **Project Browser**, right-click your project. Select **Add Source**.
- To view the include folders that you used, with your results open, select **Window** > **Show/Hide View** > **Configuration**. Under the node **Environment Settings**, you see the folders listed under **Include folders**.

## Settings

This is a read-only option available only when viewing results in the user interface of the Polyspace desktop products. Unlike other options, you do not specify include folders on the **Configuration** pane. Instead, you add your include folders on the **Project Browser** pane.

## See Also

`Include (-include)` | `-I`

# Constraint setup (-data-range-specifications)

Constrain global variables, function inputs and return values of stubbed functions

## Description

Specify constraints (also known as data range specifications or DRS) for global variables, function inputs and return values of stubbed functions using a **Constraint Specification** template file. The template file is an XML file that you can generate in the Polyspace user interface.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

**Command line and options file**: Use the option `-data-range-specifications`. See “Command-Line Information” on page 2-90.

### Why Use This Option

Use this option to constrain certain objects in your code whose values are known only at run time, for instance, user inputs or sensor values. Using correctly constrained objects might reduce false positives or orange checks in Polyspace results.

Based on your source code, Polyspace makes assumptions about items such as variable ranges and allowed buffer size for pointers. Sometimes these assumptions are broader than what you expect because:

- You have not provided the complete code. For example, you did not provide some of the function definitions.
- Some of the information about variables is available only at run time. For example, some variables in your code obtain values from the user at run time.

For instance, an `int` variable representing a real life speed cannot have a value that is smaller than zero or greater than the speed of light. Polyspace might assume that the variable has a range `[-2^32... 2^32-1]`. Because of such broad assumptions:

- Code Prover might consider more execution paths than those paths that occur at run time. If an operation fails along one of the execution paths, Polyspace places an orange check on the operation. If that execution path does not occur at run time, the orange check indicates a false positive.
- Bug Finder might produce false positives.

To reduce the number of such false positives, specify applicable constraints on global variables, function inputs, and return values of stubbed functions.

After you specify your constraints, save them as an XML file to use them for subsequent analyses. If your source code changes, update the previous constraints. Creating a new constraint template is not necessary.

## Settings

**No Default**

Enter full path to the template file. Alternately, click Edit to open a **Constraint Specification** wizard. This wizard allows you to generate a template file or navigate to an existing template file.

For more information, see “Specify External Constraints for Polyspace Analysis”.

## Command-Line Information

**Parameter:** `-data-range-specifications`
**Value:** *`file`*
**No Default**
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-data-range-specifications "C:\DRS\range.xml"`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-data-range-specifications "C:\DRS\range.xml"`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-data-range-specifications "C:\DRS\range.xml"`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-data-range-specifications "C:\DRS\range.xml"`

You can specify an absolute path to the constraints file or a path relative to the location from which you run the `polyspace-bug-finder` or `polyspace-code-prover` command.

## Examples

### Specify Range of Runtime Variable to Reduce Orange Checks

In this code, the multiplication operation in the definition of `tmp` has an orange overflow check. This check appears because Polyspace assumes that the return value of the function `getSpeed()` has a range `-2^32... 2^32-1`.

```
//file1.c
extern int getSpeed(); //Returns the reading of speedometer
int main(){
    //...
    int tmp = 2* getSpeed();
    //...
    return 0;
}
```

To resolve this orange check, constrain the return value of `getSpeed()` with a range `[0..30000000]`. Create an XML file `drs.xml` that has this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--EDRS Version 2.0-->
<global>
    <file name="C:\\Users\\example.c">
            <function name="getSpeed" line="1" attributes="extern" main_generator_called="disable
                <scalar name="return" line="1" base_type="int32" complete_type="int32" init_mode=
            </function>
    </file>
</global>
```

After specifying the data range, run the verification again. Use the command:

```
polyspace-code-prover file1.c -data-range-specifications drs.xml -lang c
```

The orange check is replaced by a green check.

```
extern int getSpeed(); //Returns the reading of speedometer
int main(){
    //...
    int tmp = 2* getSpeed();
    //...
    return 0;
}
```

## See Also

Functions to stub (-functions-to-stub) | Ignore default initialization of global variables (-no-def-init-glob)

### Topics

"Specify Polyspace Analysis Options"
"Specify External Constraints for Polyspace Analysis"

# Ignore default initialization of global variables (-no-def-init-glob)

Consider global variables as uninitialized unless explicitly initialized in code

## Description

*This option applies to Code Prover only. It does not affect a Bug Finder analysis.*

Specify that Polyspace must not consider global and static variables as initialized unless they are explicitly initialized in the code.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

**Command line and options file**: Use the option `-no-def-init-glob`. See “Command-Line Information” on page 2-93.

### Why Use This Option

The C99 Standard specifies that global variables are implicitly initialized. The default analysis follows the Standard and considers this implicit initialization.

If you want to initialize specific global variables explicitly, use this option to find the instances where global variables are not explicitly initialized.

## Settings

☑ On

Polyspace ignores implicit initialization of global and static variables. The verification generates a red **Non-initialized variable** error if your code reads a global or static variable before writing to it.

If you enable this option, global variables are considered uninitialized unless you explicitly initialize them in the code. Note that this option overrides the option `Variables to initialize (-main-generator-writes-variables)`. Even if you initialize variables with the generated `main`, this option forces the analysis to ignore the initialization.

□ Off (default)

Polyspace considers global variables and static variables to be initialized according to C99 or ISO C++ standards. For instance, the default values are:

- 0 for `int`
- 0 for `char`
- 0.0 for `float`

## Tips

Static local variables have the same lifetime as global variables even though their visibility is limited to the function where they are defined. Therefore, the option applies to static local variables.

## Command-Line Information

**Parameter:** `-no-def-init-glob`
**Default:** Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-no-def-init-glob`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-no-def-init-glob`

## See Also

`Non-initialized variable`

### Topics

"Specify Polyspace Analysis Options"

# Functions to stub (`-functions-to-stub`)

Specify functions to stub during analysis

## Description

Specify functions to stub during analysis.

For specified functions, Polyspace :

- Ignores the function definition even if it exists.
- Assumes that the function inputs and outputs have full range of values allowed by their type.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

**Command line and options file**: Use the option `-functions-to-stub`. See “Command-Line Information” on page 2-96.

### Why Use This Option

If you want the analysis to ignore the code in a function body, you can stub the function.

For instance:

- Suppose you have not completed writing the function and do not want the analysis to consider the function body. You can use this option to stub the function and then specify constraints on its return value and modifiable arguments.
- Suppose the analysis of a function body is imprecise. The analysis assumes that the function returns all possible values that the function return type allows. You can use this option to stub the function and then specify constraints on its return value.

## Settings

**No Default**

Enter function names or choose from a list.

- Click ✚ to add a field and enter the function name.
- Click 🔍 to list functions in your code. Choose functions from the list.

When entering function names, use either the basic syntax or, to differentiate overloaded functions, the argument syntax. For the argument syntax, separate function arguments with semicolons. See the following code and table for examples.

```
//simple function
```

```
void test(int a, int b);

//C++ template function

Template <class myType>
myType test(myType a, myType b);

//C++ class method

class A {
    public:
    int test(int var1, int var2);
};


//C++ template class method

template <class myType> class A
{
    public:
    myType test(myType var1, myType var2);
};
```

| **Function Type** | **Basic Syntax** | **Argument Syntax** |
|---|---|---|
| Simple function | `test` | `test(int; int)` |
| C++ template function | `test` | `test(myType; myType)` |
| C++ class method | `A::test` | `A::test(int;int)` |
| C++ template class method | `A<myType>::test` | `A<myType>::test(myType;myType)` |

## Tips

- Code Prover makes assumptions about the arguments and return values of stubbed functions. For example, Polyspace assumes that the return values of stubbed functions are full range. These assumptions can affect checks in other sections of the code. See “Code Prover Assumptions About Stubbed Functions” (Polyspace Code Prover).
- If you stub a function, you can constrain the range of function arguments and return value. To specify constraints, use the analysis option `Constraint setup (-data-range-specifications)`.
- When you use this option, you might see a change in file-level code complexity metrics such as number of lines and comment density because one or more function bodies are no longer analyzed.
- For C functions, these special characters are allowed:`( ) < > ; _`

  For C++ functions, these special characters are allowed : `( ) < > ; _ * & [ ]`

  Space characters are allowed for C++, but are not allowed for C functions.
- You cannot use this option to stub the following C++ functions:
  - `constexpr` functions
  - Function-try-blocks that associate a `catch` clause with an entire function body, for instance:

```
Class()
    try : Class( 0.0 ) //delegate constructor
    {
        // ...
    }
    catch (...)
    {
        // exception occurred on initialization
    }
```

- Template functions with a parameter pack, for instance:

```
template <class T, class... T2>
    X(T n, T n2, T2... rest): X(rest...) {
        v.insert(v.begin(), n);
        v.insert(v.begin(), n2);
    }
```

- Functions with `auto` return type, for instance:

```
template <typename F, typename... Args>
inline decltype(auto) invoke(F&& func, Args&&... args)
{
  return invoke_impl(eastl::forward<F>(func), eastl::forward<Args>(args)...);
}
```

## Command-Line Information

**Parameter:** `-functions-to-stub`
**No Default**
**Value:** *`function1`*`[,`*`function2`*`[,...]]`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-functions-to-stub function_1,function_2`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-functions-to-stub function_1,function_2`

## See Also

`Constraint setup (-data-range-specifications)`

### Topics

"Specify Polyspace Analysis Options"

# Libraries used (-library)

Specify libraries that you use in your program

## Description

Specify libraries that you use in your program.

The analysis uses smart stubs for functions from those libraries instead of generic stubs and does not attempt to check the function implementations. Using this option enables faster analysis without losing precision and triggers library-specific checks on function calls.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

**Command line and options file**: Use the option `-library`. See "Command-Line Information" on page 2-98.

### Why Use This Option

For faster and library-aware analysis, use this option. Unless you use this option, the analysis either attempts to check the library implementation or if the implementation is not available, uses generic stubs for library functions. Checking the function bodies can increase analysis time significantly while using generic stubs can lead to loss of precision.

The option also triggers library-specific checks on function arguments. For instance, if you select the option value `autosar`, a Bug Finder or Code Prover analysis checks arguments to functions from the AUTOSAR RTE API for compliance with the AUTOSAR Standard.

## Settings

**Default:** `none`

`none`

The analysis uses smart stubs only for functions from the C Standard Library.

`autosar`

The analysis uses smart stubs for AUTOSAR RTE API functions even if their implementations are available.

The option also triggers AUTOSAR-specific checks on function arguments. For more information, see the corresponding checkers:

- Bug Finder: `Non-compliance with AUTOSAR specification`

  Besides setting the option, you must also explicitly enable the above checker (or enable all checkers).
- Code Prover: `Non-compliance with AUTOSAR specification`

Setting the option is sufficient to enable the checker.

`stdlibcxx`

The analysis uses smart stubs for methods from C++ Standard Library containers even if their implementations are available.

The containers whose methods are stubbed include `std::map`, `std::unordered_map`, `std::deque`, `std::vector`, `std::set`, `std::unordered_set`, and `std::list`. If you use this option and your code contains C++ container methods, the analysis log contains a message:

```
n methods stubbed for better performance
```

where *n* is the number of methods stubbed.

Note that unlike functions from the C Standard Library, the stubs of C++ container methods are not used to check for domain errors and other error conditions. The methods are stubbed to avoid the unnecessary performance costs from analyzing their implementation. As a result, if you invoke a C++ container method with incorrect arguments, you do not see errors that refer to the incorrect invocation. But in some cases, you might see other errors that indirectly follow from the incorrect invocation. For instance, if you invoke the `std::stack::top()` method on an empty stack, you see a `Non-initialized variable` error even though the stack itself is initialized.

This option has effect only on a Code Prover analysis. A Bug Finder analysis discards implementations of C++ container methods immediately after compilation and does not require the smart stubbing.

### Command-Line Information

**Parameter:** `-library`
**No Default**
**Value:** `autosar | stdlibcxx`
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-library autosar -checkers autosar_lib_non_compliance`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-library autosar`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-library autosar -checkers autosar_lib_non_compliance`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-library autosar`

## Version History

**Introduced in R2021a**

### See Also

**Topics**

"Specify Polyspace Analysis Options"

# Generate stubs for Embedded Coder lookup tables (`-stub-embedded-coder-lookup-table-functions`)

Stub autogenerated functions that use lookup tables and model them more precisely

## Description

*This option is available only for model-generated code. The option is relevant only if you generate code from a Simulink® model that uses Lookup Table blocks using MathWorks code generation products.*

Specify that the verification must stub autogenerated functions that use certain kinds of lookup tables in their body. The lookup tables in these functions use linear interpolation and do not allow extrapolation. That is, the result of using the lookup table always lies between the lower and upper bounds of the table.

### Set Option

If you are running verification from Simulink, use the option “Stub lookup tables” (Polyspace Code Prover) in Simulink Configuration Parameters, which performs the same task.

**User interface** (desktop products only): In your Polyspace project configuration, the option is on the **Inputs & Stubbing** node.

**Command line and options file**: Use the option `-stub-embedded-coder-lookup-table-functions`. See “Command-Line Information” on page 2-100.

### Why Use This Option

If you use this option, the verification is more precise and has fewer orange checks. The verification of lookup table functions is usually imprecise. The software has to make certain assumptions about these functions. To avoid missing a run-time error, the verification assumes that the result of using the lookup table is within the full range allowed by the result data type. This assumption can cause many unproven results (orange checks) when a lookup table function is called. By using this option, you narrow down the assumption. For functions that use lookup tables with linear interpolation and no extrapolation, the result is at least within the bounds of the table.

The option is relevant only if your model has Lookup Table blocks. In the generated code, the functions corresponding to Lookup Table blocks also use lookup tables. The function names follow specific conventions. The verification uses the naming conventions to identify if the lookup tables in the functions use linear interpolation and no extrapolation. The verification then replaces such functions with stubs for more precise verification.

## Settings

☑ On (default)

For autogenerated functions that use lookup tables with linear interpolation and no extrapolation, the verification:

- Does not check for run-time errors in the function body.
- Calls a function stub instead of the actual function at the function call sites. The stub ensures that the result of using the lookup table is within the bounds of the table.

To identify if the lookup table in the function uses linear interpolation and no extrapolation, the verification uses the function name. In your analysis results, you see that the function is not analyzed. If you place your cursor on the function name, you see the following message:

```
Function has been recognized as an Embedded Coder Lookup-Table function.
It was stubbed by Polyspace to increase precision.
Unset the -stub-embedded-coder-lookup-table-functions option to analyze
   the code below.
```

□ Off

The verification does not stub autogenerated functions that use lookup tables.

## Tips

- The option applies to only autogenerated functions. If you integrate your own C/C++ S-Function using lookup tables with the model, these functions do not follow the naming conventions for autogenerated functions. The option does not cause them to be stubbed. If you want the same behavior for your handwritten lookup table functions as the autogenerated functions, use the option `-code-behavior-specifications` and map your function to the `__ps_lookup_table_clip` function.
- If you run verification from Simulink, the option is on by default. For certification purposes, if you want your verification tool to be independent of the code generation tool, turn off the option.

## Command-Line Information

**Parameter:** `-stub-embedded-coder-lookup-table-functions`
**Default**: On
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-stub-embedded-coder-lookup-table-functions`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-stub-embedded-coder-lookup-table-functions`

# Version History

**Introduced in R2016b**

## See Also

### Topics

"Specify Polyspace Analysis Options"

# Generate results for sources and (`-generate-results-for`)

Specify files on which you want analysis results

## Description

*This option affects a Bug Finder analysis only.*

Specify files on which you want analysis results.

By default, results appear on source files and header files in the same folder as the source files. You can use this option to see results in other header files. If you use the option `Do not generate results for (-do-not-generate-results-for)` to suppress entire folders, you can use this option to unsuppress some subfolders or files in those folders.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

**Command line and options file**: Use the option `-generate-results-for`. See "Command-Line Information" on page 2-102.

### Why Use This Option

Use this option to see results in header files that are most relevant to you.

For instance, by default, results are generated on header files that are located in the same folder as the source files. Often, other header files belong to a third-party library. Though these header files are required for a precise analysis, you are not interested in reviewing findings in those headers. Therefore, by default, results are not generated for those headers. If you *are interested* in certain headers from third-party libraries, change the default value of this option.

Note that in Polyspace as You Code, you cannot see results in headers `#include`-d through a source file *at all*. The default behavior is to consider the headers in the same folder as the source file (or subfolders) for analysis but suppress results found in the headers. You can use this option only to expand the scope of which headers are considered during analysis. See also "Analysis Scope of Polyspace as You Code".

## Settings

**Default**: `source-headers`

`source-headers`

Results appear on source files and header files in the same folder as the source files or in subfolders of source file folders.

The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument `-sources` at the command line).

`all-headers`

Results appear on source files and all header files. The header files can be in the same folder as source files, in subfolders of source file folders or in include folders.

The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument `-sources` at the command line).

The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument `-I` at the command line).

`custom`

Results appear on source files and the files that you specify. If you enter a folder name, results appear on header files in that folder (and its subfolders).

Click ✚ to add a field. Enter a file or folder name.

## Tips

1 Use this option in combination with appropriate values for the option `Do not generate results for (-do-not-generate-results-for)`.

If you choose `custom` and the values for the two options conflict, the more specific value determines the display of results. For instance, in the following examples, the value for the option **Generate results for sources and** is more specific.

| Generate results for sources and | Do not generate results for | Final Result |
|---|---|---|
| `custom`:<br>`C:\Includes\Custom_Library\` | `custom`:<br>`C:\Includes` | Results are displayed on header files in `C:\Includes\Custom_Library\` and its subfolders but not generated for other header files in `C:\Includes`. |
| `custom`:<br>`C:\Includes\my_header.h` | `custom`:<br>`C:\Includes\` | Results are displayed on the header file `my_header.h` in `C:\Includes\` but not generated for other header files in `C:\Includes\` and its subfolders. |

Using these two options together, you can suppress results from all files in a certain folder but unsuppress select files in those folders.

2 If you choose `all-headers` for this option, results are displayed on all header files irrespective of what you specify for the option **Do not generate results for**.

## Command-Line Information

**Parameter:** `-generate-results-for`
**Value:** `source-headers` | `all-headers` | `custom=`*file1*`[,`*file2*`[,...]]` | `custom=`*folder1*`[,`*folder2*`[,...]]`

**Example (Bug Finder):** `polyspace-bug-finder -lang c -sources` *`file_name`* `-misra2 required-rules -generate-results-for custom="C:\usr\include"`
**Example (Code Prover):** `polyspace-code-prover -lang c -sources` *`file_name`* `-misra2 required-rules -generate-results-for custom="C:\usr\include"`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -lang c -sources` *`file_name`* `-misra2 required-rules -generate-results-for custom="C:\usr\include"`
**Example (Code Prover Server):** `polyspace-code-prover-server -lang c -sources` *`file_name`* `-misra2 required-rules -generate-results-for custom="C:\usr\include"`

## Version History

**Introduced in R2016a**

### See Also

**Topics**
"Specify Polyspace Analysis Options"

# Do not generate results for (`-do-not-generate-results-for`)

Specify files on which you do not want analysis results

## Description

*This option affects a Bug Finder analysis only.*

Specify files on which you do not want analysis results.

By default, results do not appear on header files (unless they are in the same folder as the source files). You can use this option to suppress results from some source files too (or from header files in the same folders as source files). If you use the option `Generate results for sources and (-generate-results-for)` to show results on some include folders, you can use this option to suppress results from some subfolders or files in those include folders.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node.

**Command line and options file**: Use the option `-do-not-generate-results-for`. See "Command-Line Information" on page 2-107.

### Why Use This Option

Use this option to see results in header files that are most relevant to you.

For instance, by default, results are generated on header files that are located in the same folder as the source files. If you are not interested in reviewing the findings in those headers, change the default value of this option.

Note that in Polyspace as You Code, the default behavior is to not even analyze the headers in non-source folders. You can use this option to expand the scope of not analyzed files to all headers or a different subset of headers. See also "Analysis Scope of Polyspace as You Code".

## Settings

**Default**: `include-folders`

`include-folders`

Results are not generated for header files in include folders (and their subfolders).

The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument `-I` at the command line).

If an include folder is a subfolder of a source folder, results are generated for files in that include folder even if you specify the option value `include-folders`. In this situation, use the option value `custom` and explicitly specify the include folders to ignore.

`all-headers`

Results are not generated for all header files. The header files can be in the same folder as source files, in subfolders of source file folders or in include folders.

The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument `-sources` at the command line).

The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument `-I` at the command line).

`custom`

Results are not generated for the files that you specify. If you enter a folder name, results are suppressed from files in that folder (and its subfolders).

Click ✚ to add a field. Enter a file or folder name.

## Tips

1 Use this option appropriately in combination with appropriate values for the option `Generate results for sources and (-generate-results-for)`.

If you choose `custom` and the values for the two options conflict, the more specific value determines the display of results. For instance, in the following examples, the value for the option **Generate results for sources and** is more specific.

| Generate results for sources and | Do not generate results for | Final Result |
|---|---|---|
| `custom`:<br>`C:\Includes\Custom_Library\` | `custom`:<br>`C:\Includes` | Results are displayed on header files in `C:\Includes\Custom_Library\` and its subfolders but not generated for other header files in `C:\Includes`. |
| `custom`:<br>`C:\Includes\my_header.h` | `custom`:<br>`C:\Includes\` | Results are displayed on the header file `my_header.h` in `C:\Includes\` but not generated for other header files in `C:\Includes\` and its subfolders. |

Using these two options together, you can suppress results from all files in a certain folder but unsuppress select files in those folders.

2 If you choose `all-headers` for this option, results are suppressed from all header files irrespective of what you specify for the option **Generate results for sources and**.

3 If a defect or coding rule violation involves two files and you do not generate results for one of the files, the defect or rule violation still appears. For instance, if you define two variables with similar-looking names in files `myFile.cpp` and `myFile.h`, you get a violation of the MISRA™ C ++ rule 2-10-1, even if you do not generate results for `myFile.h`. MISRA C++ rule 2-10-1 states that different identifiers must be typographically unambiguous.

The following results can involve more than one file:

### MISRA C: 2004 Rules

- MISRA C: 2004 Rule 5.1 — Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
- MISRA C: 2004 Rule 5.2 — Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
- MISRA C: 2004 Rule 8.8 — An external object or function shall be declared in one file and only one file.
- MISRA C: 2004 Rule 8.9 — An identifier with external linkage shall have exactly one external definition.

### MISRA C: 2012 Directives and Rules

- MISRA C: 2012 Directive 4.5 — Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
- MISRA C: 2012 Rule 5.2 — Identifiers declared in the same scope and name space shall be distinct.
- MISRA C: 2012 Rule 5.3 — An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
- MISRA C: 2012 Rule 5.4 — Macro identifiers shall be distinct.
- MISRA C: 2012 Rule 5.5 — Identifiers shall be distinct from macro names.
- MISRA C: 2012 Rule 8.5 — An external object or function shall be declared once in one and only one file.
- MISRA C: 2012 Rule 8.6 — An identifier with external linkage shall have exactly one external definition.

### MISRA C++ Rules

- MISRA C++ Rule 2-10-1 — Different identifiers shall be typographically unambiguous.
- MISRA C++ Rule 2-10-2 — Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
- MISRA C++ Rule 3-2-2 — The One Definition Rule shall not be violated.
- MISRA C++ Rule 3-2-3 — A type, object or function that is used in multiple translation units shall be declared in one and only one file.
- MISRA C++ Rule 3-2-4 — An identifier with external linkage shall have exactly one definition.
- MISRA C++ Rule 7-5-4 — Functions should not call themselves, either directly or indirectly.
- MISRA C++ Rule 15-4-1 — If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.

### JSF C++ Rules

- JSF C++ Rule 46 — User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.
- JSF C++ Rule 48 — Identifiers will not differ by only a mixture of case, the presence/absence of the underscore character, the interchange of the letter `O` with the number 0 or the letter `D`, the interchange of the letter `I` with the number 1 or the letter `l`, the interchange of the letter `S` with the number 5, the interchange of the letter `Z` with the number 2 and the interchange of the letter `n` with the letter `h`.

- JSF C++ Rule 137 — All declarations at file scope should be static where possible.
- JSF C++ Rule 139 — External objects will not be declared in more than one file.

**Polyspace Bug Finder Defects**

- `Variable shadowing` — Variable hides another variable of same name with nested scope.
- `Declaration mismatch` — Mismatch occurs between function or variable declarations.

4 If a result (coding rule violation or Bug Finder defect) is inside a macro, Polyspace typically shows the result on the macro definition instead of the macro occurrences so that you review the result only once. Even if the macro is used in a suppressed file, the result is still shown on the macro definition, *if the definition occurs in an unsuppressed file*.

## Command-Line Information

**Parameter:** `-do-not-generate-results-for`
**Value:** `all-headers` | `include-folders` | `custom=`*`file1`*`[,`*`file2`*`[,...]]` | `custom=`*`folder1`*`[,`*`folder2`*`[,...]]`
**Example (Bug Finder):** `polyspace-bug-finder -lang c -sources` *`file_name`* `-misra2 required-rules -do-not-generate-results-for custom="C:\usr\include"`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -lang c -sources` *`file_name`* `-misra2 required-rules -do-not-generate-results-for custom="C:\usr\include"`

# Version History

**Introduced in R2016a**

## See Also

`Generate results for sources and (-generate-results-for)`

### Topics

“Specify Polyspace Analysis Options”

# External multitasking configuration

Enable setup of multitasking configuration from external file definitions

## Description

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify whether you want to use definitions from external files to set up the multitasking configuration of your Polyspace project. The supported external file formats are:

- ARXML files for AUTOSAR projects
- OIL files for OSEK projects

### Set Option

**User interface:** In the **Configuration** pane, the option is available on the **Multitasking** node.

**Command line and options file**: See "Command-Line Information" on page 2-108.

### Why Use This Option

If your AUTOSAR project includes ARXML files with ECU configuration parameters, or if your OSEK project includes OIL files, Polyspace can parse these files. The software sets up tasks, interrupts, cyclical tasks, and critical sections. You do not have to set them up manually.

## Settings

☑ On

Polyspace parses the external files that you provide in the format that you specify to set up the multitasking configuration of your project.

`osek`

Look for and parse OIL files to extract multitasking description.

`autosar`

Look for and parse AUTOSAR XML files to extract multitasking description.

□ Off (default)

Polyspace does not set up the multitasking configuration of your project.

## Command-Line Information

There is no single command-line option to turn on external multitasking configuration. By using the `-osek-multitasking` option or the `-autosar-multitasking` option, you enable external multitasking configuration.

# Version History

**Introduced in R2018a**

## See Also

`ARXML files selection (-autosar-multitasking)` | `OIL files selection (-osek-multitasking)`

### Topics

"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"

# OIL files selection (`-osek-multitasking`)

Set up multitasking configuration from OIL file definition

## Description

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify the OIL files that Polyspace parses to set up the multitasking configuration of your OSEK project.

### Set Option

**User interface:** In the **Configuration** pane, the option is available on the **Multitasking** node. See Dependencies on page 2-114 for other options you must also enable.

**Command line: and options file** Use the option `-osek-multitasking`. See “Command-Line Information” on page 2-114.

### Why Use This Option

If your project includes OIL files, Polyspace can parse these files to set up tasks, interrupts, cyclical tasks, and critical sections. You do not have to set them up manually.

## Settings

☑ On

Polyspace looks for and parses OIL files to set up your multitasking configuration.

`auto`

Look for OIL files in your project source and include folders, but not in their subfolders.

`custom`

Look for OIL files on the specified path and the path subfolders. You can specify a path to the OIL files or to the folder containing the files.

When you select this option, in your source code, Polyspace supports these OSEK multitasking keywords:

- `TASK`
- `DeclareTask`
- `ActivateTask`
- `DeclareResource`
- `GetResource`
- `ReleaseResource`
- `ISR`
- `DeclareEvent`

- DeclareAlarm

Polyspace parses the OIL files that you provide for TASK, ISR, RESOURCE, and ALARM definitions. The analysis uses these definitions and the supported multitasking keywords to configure tasks, interrupts, cyclical tasks, and critical sections.

### Example: Analyze Your OSEK Multitasking Project

This example shows how to set up the multitasking configuration of an OSEK project and run an analysis on this project. To try the steps in this example, use the demo files in the folder *polyspaceroot*/help/toolbox/bugfinder/examples/External_multitasking/OSEK or *polyspaceroot*/help/toolbox/codeprover/examples/External_multitasking/OSEK. *polyspaceroot* is the Polyspace installation folder. The analysis results apply to this example code.

```
#include <assert.h>
#include "include/example_osek_multi.h"

int var1;
int var2;
int var3;

DeclareAlarm(Cyclic_task_activate);
DeclareResource(res1);
DeclareTask(init);
TASK(afterinit1);

TASK(init) // task
{


  var2++;
  ActivateTask(afterinit1);
  var3++;
  GetResource(res1); // critical section begins
  var1++;
  ReleaseResource(res1); // critical section ends
}

TASK(afterinit1) // task
{
  var3++;
  var2++;
  GetResource(res1); // critical section begins
  var1++;
  ReleaseResource(res1); // critical section ends

}
int var4;
void func()
{
  var4++;
}

TASK(Cyclic_task) // cyclic task
{
  func();
}

void main()
{}
```

To set up your multitasking configuration and analyze the code:

1. Copy the contents of *polyspaceroot*/help/toolbox/bugfinder/examples/External_multitasking/OSEK or *polyspaceroot*/help/toolbox/codeprover/examples/External_multitasking/OSEK to your machine, for instance in C:\Polyspace_worskpace\OSEK.
2. Run an analysis on your OSEK project by using the command:

   - Bug Finder:

```
polyspace-bug-finder -sources ^
C:\Polyspace_workspace\OSEK\example_osek_multitasking.c ^
-osek-multitasking auto
```

- Code Prover:

  ```
  polyspace-code-prover -sources ^
  C:\Polyspace_workspace\OSEK\example_osek_multitasking.c ^
  -osek-multitasking auto
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -sources ^
  C:\Polyspace_workspace\OSEK\example_osek_multitasking.c ^
  -osek-multitasking auto
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -sources ^
  C:\Polyspace_workspace\OSEK\example_osek_multitasking.c ^
  -osek-multitasking auto
  ```

Bug Finder detects a data race on variable `var3` because of multiple read and write operation from tasks `init` and `afterinit1`. See `Data race`.

```
#include <assert.h>
#include "include/example_osek_multi.h"

int var1;
int var2;
int var3;
```

There is no defect on `var2` since `afterinit1` goes to an active state (`ActivateTask()`) after `init` increments `var2`. Similarly, there is no defect on `var1` because it is protected by the `GetResource()` and `ReleaseResource()` calls.

Code Prover detects that `var3` is a potentially unprotected global variable because it is used in tasks `init` and `afterinit1` with no protection from interruption during the read and write operations. The analysis also shows that the cyclic task operation on `var4` can potentially cause an overflow. See `Potentially unprotected variable` and `Overflow`.

```
#include <assert.h>
#include "include/example_osek_multi.h"

int var1;
int var2;
int var3;

...
void func()
{
  var4++;
}
```

Variable `var2` is not shared because `afterinit1` goes to an active state (`ActivateTask()`) after `init` increments `var2`. Variable `var1` is a protected variable (Polyspace Code Prover) through the critical sections from the `GetResource()` and `ReleaseResource()` calls.

To see how Polyspace models the `TASK`, `ISR`, and `RESOURCE` definitions from your OIL files, open the **Concurrency window** from the **Dashboard** pane.

□ Off (default)

Polyspace does not set up a multitasking configuration for your OSEK project.

#### Additional Considerations

- Make sure that you declare all tasks by using the `DeclareTask` or `TASK` keywords before you pass those tasks as parameters to functions or macros that expect a task. For example , if you pass task `foo` to `ActivateTask` without using `DeclareTask(foo);` first, Polyspace considers task `foo` undefined which results in a compilation error.
- The analysis ignores `TerminateTask()` declarations in your source code and considers that subsequent code is executed.
- Polyspace ignores syntax elements of your OIL files that do not follow the syntax defined here.

### Dependencies

To enable this option in the user interface of the desktop products, first select the option `External multitasking configuration`.

### Command-Line Information

**Parameter:** `-osek-multitasking`
**Value:** `auto` | `custom='`*`file1 [,file2, dir1,...]`*`'`
**Default:** Off
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`source_path`* `-I` *`include_path`* `-osek-multitasking custom='path\to\file1.oil, path\to\dir'`
**Example (Code Prover):** `polyspace-code-prover -sources` *`source_path`* `-I` *`include_path`* `-osek-multitasking custom='path\to\file1.oil, path\to\dir'`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`source_path`* `-I` *`include_path`* `-osek-multitasking custom='path\to\file1.oil, path\to\dir'`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`source_path`* `-I` *`include_path`* `-osek-multitasking custom='path\to\file1.oil, path\to\dir'`

## Version History

**Introduced in R2017b**

### See Also

`Show global variable sharing and usage only (-shared-variables-mode)`

#### Topics

"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"

# ARXML files selection (-autosar-multitasking)

Set up multitasking configuration from ARXML file definitions

## Description

*To detect data races in large AUTOSAR applications, use this option with Polyspace Bug Finder.*

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify the ARXML files that Polyspace parses to set up the multitasking configuration of your AUTOSAR project.

### Set Option

**User interface:** In the **Configuration** pane, the option is available on the **Multitasking** node. See Dependencies on page 2-116 for other options you must also enable.

**Command line: and options file** Use the option `-autosar-multitasking`. See “Command-Line Information” on page 2-114.

### Why Use This Option

If your project includes ARXML files with `<ECUC-CONTAINER-VALUE>` elements, Polyspace can parse these files to set up tasks, interrupts, cyclical tasks, and critical sections. You do not have to set them up manually.

## Settings

☑ On

Polyspace looks for and parses ARXML files to set up your multitasking configuration.

When you select this option, the software assumes that you use the OSEK multitasking API in your source code to declare and define tasks and interrupts. Polyspace supports these OSEK multitasking keywords:

- `TASK`
- `DeclareTask`
- `ActivateTask`
- `DeclareResource`
- `GetResource`
- `ReleaseResource`
- `ISR`
- `DeclareEvent`
- `DeclareAlarm`

Polyspace parses the ARXML files that you provide for `OsTask`, `OsIsr`, `OsResource`, `OsAlarm`, and `OsEvent` definitions. The analysis uses these definitions and the supported multitasking keywords to configure tasks, interrupts, cyclical tasks, and critical sections.

To see how Polyspace models the `OsTask`, `OsIsr`, and `OsResource` definitions from your ARXML files, open the **Concurrency window** from the **Dashboard** pane. In that window, under the **Entry points** column, the names of the elements are extracted from their `<SHORT-NAME>` values in the ARXML files.

☐ Off (default)

Polyspace does not set up a multitasking configuration for your AUTOSAR project.

### Additional Considerations

- The analysis ignores `TerminateTask()` declarations in your source code and considers that subsequent code is executed.
- Polyspace supports multitasking configuration only from ARXML files for AUTOSAR specification version 4.0 and later.

## Dependencies

To enable this option in the user interface of the desktop products, first select the option `External multitasking configuration`.

## Command-Line Information

**Parameter:** `-autosar-multitasking`
**Value:** `file1 [,file2, dir1,...]`
**Default:** Off
**Example (Bug Finder):** `polyspace-bug-finder -sources source_path -I include_path -autosar-multitasking C:\Polyspace_Workspace\AUTOSAR\myFile.arxml`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources source_path -I include_path -autosar-multitasking C:\Polyspace_Workspace\AUTOSAR\myFile.arxml`

# Version History

**Introduced in R2018a**

## See Also

`External multitasking configuration` | `OIL files selection (-osek-multitasking)` | `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)` | `Show global variable sharing and usage only (-shared-variables-mode)`

### Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”

# Configure multitasking manually

Consider that code is intended for multitasking

## Description

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify whether your code is a multitasking application. This option allows you to manually configure the multitasking structure for Polyspace.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Multitasking** node.

**Command line and options file**: See “Command-Line Information” on page 2-118.

### Why Use This Option

By default, Bug Finder determines your multitasking model from your use of multithreading functions. In Code Prover, you have to enable automatic concurrency detection with the option `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`. However, in some cases, using automatic concurrency detection can slow down the Code Prover analysis.

In cases where automatic concurrency detection is not supported, you can explicitly specify your multitasking model by using this option. Once you select this option, you can explicitly specify your entry point functions, cyclic tasks, interrupts and protection mechanisms for shared variables, such as critical section details.

A Code Prover verification uses your specifications to determine:

- Whether a global variable is shared.

  See “Global Variables” (Polyspace Code Prover).
- Whether a run-time error can occur.

  For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For more information, see “Concurrency Defects”.

## Settings

☑ On

The code is intended for a multitasking application.

You have to explicitly specify your multitasking configuration using other Polyspace options. See “Configuring Polyspace Multitasking Analysis Manually”.

□ Off (default)

The code is not intended for a multitasking application.

Disabling the option has this additional effect in Code Prover:

- If a `main` exists, Code Prover verifies only those functions that are called by the `main`.
- If a `main` does not exist, Polyspace verifies the functions that you specify. To verify the functions, Polyspace generates a `main` function and calls functions from the generated `main` in a sequence that you specify. For more information, see `Verify module or library (-main-generator)`.

## Tips

If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

## Command-Line Information

There is no single command-line option to turn on multitasking analysis. By using any of the options `Tasks (-entry-points)`, `Cyclic tasks (-cyclic-tasks)` or `Interrupts (-interrupts)`, you turn on multitasking analysis.

## See Also

`-preemptable-interrupts` | `-non-preemptable-tasks` | `Tasks (-entry-points)` | `Cyclic tasks (-cyclic-tasks)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

### Topics

"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Configuring Polyspace Multitasking Analysis Manually"
"Protections for Shared Variables in Multitasking Code"

# Enable automatic concurrency detection for Code Prover (`-enable-concurrency-detection`)

Automatically detect certain families of multithreading functions

## Description

*This option affects a Code Prover analysis only.*

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify whether the analysis must automatically detect POSIX®, VxWorks®, Windows, µC/OS II and other multithreading functions.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” (Polyspace Code Prover) for other options that you must enable or disable.

**Command line and options file**: Use the option `-enable-concurrency-detection`. See “Command-Line Information” on page 2-120.

### Why Use This Option

If you use this option, Polyspace determines your multitasking model from your use of multithreading functions. In Bug Finder, automatic concurrency detection is enabled by default. In Code Prover, you have to explicitly enable automatic concurrency detection.

In some cases, using automatic concurrency detection can slow down the Code Prover analysis. In those cases, you can choose to not enable this option and explicitly specify your multitasking model. See “Configuring Polyspace Multitasking Analysis Manually”.

## Settings

☑ On

If you use one of the supported functions for multitasking, the analysis automatically detects your multitasking model from your code.

For a list of supported multitasking functions and limitations in auto-detection of threads, see “Auto-Detection of Thread Creation and Critical Section in Polyspace”.

□ Off (default)

The analysis does not attempt to detect the multitasking model from your code.

If you want to manually configure your multitasking model, see “Configuring Polyspace Multitasking Analysis Manually”.

## Dependencies

If you enable this option, your code must contain a `main` function. You cannot use the Code Prover options to generate a `main`.

## Command-Line Information

**Parameter:** `-enable-concurrency-detection`
**Default:** Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-enable-concurrency-detection`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-enable-concurrency-detection`

## See Also

`Show global variable sharing and usage only (-shared-variables-mode)`

### Topics

"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Auto-Detection of Thread Creation and Critical Section in Polyspace"

# Tasks (`-entry-points`)

Specify functions that serve as tasks to your multitasking application

## Description

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify functions that serve as tasks to your code. If the function does not exist, the verification warns you and continues the verification.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 2-122 for other options you must also enable.

**Command line and options file**: Use the option `-entry-points`. See “Command-Line Information” on page 2-122.

### Why Use This Option

Use this option when your code is intended for multitasking.

To specify cyclic tasks and interrupts, use the options `Cyclic tasks (-cyclic-tasks)` and `Interrupts (-interrupts)`. Use this option to specify other tasks.

A Code Prover analysis uses your specifications to determine:

- Whether a global variable is shared.

  See “Global Variables” (Polyspace Code Prover).
- Whether a run-time error can occur.

  For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For more information, see “Concurrency Defects”.

## Settings

**No Default**

Enter function names or choose from a list.

- Click ✚ to add a field and enter the function name.
- Click 🔍 to list functions in your code. Choose functions from the list.

## Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

## Tips

- In Code Prover, the functions representing entry points must have the form

  ```
  void functionName (void)
  ```
- If a function `func` takes arguments or returns a value, you cannot use it directly as an entry point. To use `func` as an entry point:, call `func` from a wrapper `void-void` function and specify the wrapper as an entry point. See “Configuring Polyspace Multitasking Analysis Manually”.
- If you specify a function as a task, you must provide its definition. Otherwise, a Code Prover verification stops with the error message:

  ```
  task func_name must be a userdef function without parameters
  ```

  A Bug Finder analysis continues but does not consider the function as an entry point.
- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.
- The Polyspace multitasking analysis assumes that a task cannot interrupt itself.

## Command-Line Information

**Parameter:** `-entry-points`
**No Default**
**Value:** `*function1*[,*function2*[,...]]`
**Example (Bug Finder):** `polyspace-bug-finder -sources *file_name* -entry-points func_1,func_2`
**Example (Code Prover):** `polyspace-code-prover -sources *file_name* -entry-points func_1,func_2`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources *file_name* -entry-points func_1,func_2`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources *file_name* -entry-points func_1,func_2`

## See Also

`Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `-preemptable-interrupts` | `-non-preemptable-tasks` | `Show global variable sharing and usage only (-shared-variables-mode)`

### Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”

# Cyclic tasks (-cyclic-tasks)

Specify functions that represent cyclic tasks

## Description

*The option is not available for code generated from MATLAB code or Simulink models.*

Specify functions that represent cyclic tasks. The analysis assumes that operations in the function body:

- Can execute any number of times.
- Can be interrupted by noncyclic tasks, other cyclic tasks and interrupts. Noncyclic tasks are specified with the option `Tasks (-entry-points)` and interrupts are specified with the option `Interrupts (-interrupts)`.

  To model a cyclic task that cannot be interrupted by other cyclic tasks, specify the task as nonpreemptable. See `-non-preemptable-tasks`. For examples, see “Define Task Priorities for Data Race Detection in Polyspace”.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 2-124 for other options you must also enable.

**Command line and options file**: Use the option `-cyclic-tasks`. See “Command-Line Information” on page 2-125.

### Why Use This Option

Use this option to specify cyclic tasks in your multitasking code. The functions that you specify must have the prototype:

```
void function_name(void);
```

A Bug Finder analysis uses your specifications to look for concurrency defects. For the `Data race` defect, the software establishes the following relations between preemptable tasks and other tasks.

- *Data race between two preemptable tasks*:

  Unless protected, two operations in different preemptable tasks can interfere with each other. If the operations use the same shared variable without protection, a data race can occur.

  If both operations are atomic, to see the defect, you have to enable the checker **Data race including atomic operations**.
- *Data race between a preemptable task and a nonpreemptable task or interrupt*:
  - An atomic operation in a preemptable task cannot interfere with an operation in a nonpreemptable task or an interrupt. Even if the operations use the same shared variable without protection, a data race cannot occur.

  - A nonatomic operation in a preemptable task also cannot interfere with an operation in a nonpreemptable task or an interrupt. However, the latter operation can interrupt the former. Therefore, if the operations use the same shared variable without protection, a data race can occur.

For more information, see:

- “Define Task Priorities for Data Race Detection in Polyspace”
- “Concurrency Defects”

A Code Prover verification uses your specifications to determine:

- Whether a global variable is shared.

  See “Global Variables” (Polyspace Code Prover).
- Whether a run-time error can occur.

  For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

## Settings

**No Default**

Enter function names or choose from a list.

- Click ✚ to add a field and enter the function name.
- Click 🔍 to list functions in your code. Choose functions from the list.

## Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

## Tips

- In Code Prover, the functions representing cyclic tasks must have the form

  ```
  void functionName (void)
  ```
- If a function `func` takes arguments or returns a value, you cannot use it directly as a cyclic task. To use `func` as a cyclic task:, call `func` from a wrapper `void-void` function and specify the wrapper as a cyclic task. See “Configuring Polyspace Multitasking Analysis Manually”.
- If you specify a function as a cyclic task, you must provide its definition. Otherwise, a Code Prover verification stops with the error message:

  ```
  task func_name must be a userdef function without parameters
  ```

  A Bug Finder analysis continues but does not consider the function as a cyclic task.
- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

- The Polyspace multitasking analysis assumes that a task cannot interrupt itself.
- Code Prover interprets this option with some limitations. The reason is that Code Prover considers all operations as potentially non-atomic and interruptible. This overapproximation leads to situations where the option might appear to be ignored. For an example, see "Effect of Task Priorities in Code Prover" (Polyspace Code Prover).

### Command-Line Information

**Parameter:** `-cyclic-tasks`
**No Default**
**Value:** `*function1*[,*function2*[,...]]`
**Example (Bug Finder):** `polyspace-bug-finder -sources *file_name* -cyclic-tasks func_1,func_2`
**Example (Code Prover):** `polyspace-code-prover -sources *file_name* -cyclic-tasks func_1,func_2`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources *file_name* -cyclic-tasks func_1,func_2`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources *file_name* -cyclic-tasks func_1,func_2`

## Version History

**Introduced in R2016b**

### See Also

`-preemptable-interrupts` | `-non-preemptable-tasks` | `Interrupts (-interrupts)` | `Tasks (-entry-points)` | `Show global variable sharing and usage only (-shared-variables-mode)`

**Topics**

"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Configuring Polyspace Multitasking Analysis Manually"
"Protections for Shared Variables in Multitasking Code"
"Define Task Priorities for Data Race Detection in Polyspace"

# Interrupts (-interrupts)

Specify functions that represent nonpreemptable interrupts

## Description

*The option is not available for code generated from MATLAB code or Simulink models.*

Specify functions that represent nonpreemptable interrupts. The analysis assumes that operations in the function body:

- Can execute any number of times.
- Cannot be interrupted by noncyclic tasks, cyclic tasks or other interrupts. Noncyclic tasks are specified with the option `Tasks (-entry-points)` and cyclic tasks are specified with the option `Cyclic tasks (-cyclic-tasks)`.

  You can also make interrupts preemptable.

  - To model an interrupt that can be interrupted by other interrupts, specify the interrupt as preemptable. See `-preemptable-interrupts`. For examples, see “Define Task Priorities for Data Race Detection in Polyspace”.
  - To make only a section of an interrupt preemptable, call a routine enabling all interrupts before that section and call another routine disabling all interrupts after the section is complete. For instance, if you specify the routine `isr()` as an interrupt, it is nonpreemptable by default. However, within `isr()`, if you call a routine enabling all interrupts, the section following the call is preemptable till you call another routine disabling all interrupts:

    ```
    void isr() {
       x++; //Nonpreemptable
       enable_all_interrupts(); //Routine enabling interrupts
       y++; //Preemptable
       disable_all_interrupts(); //Routine disabling interrupts
       z++; //Nonpreemptable
    }
    ```

    For information on how to enable and disable all interrupts, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 2-127 for other options you must also enable.

**Command line and options file**: Use the option `-interrupts`. See “Command-Line Information” on page 2-128.

### Why Use This Option

Use this option to specify interrupts in your multitasking code. The functions that you specify must have the prototype:

```
void function_name(void);
```

A Bug Finder analysis uses your specifications to look for concurrency defects. For the `Data race` defect, the analysis establishes the following relations between interrupts and other tasks:

- *Data race between two interrupts*:

  Two operations in different interrupts cannot interfere with each other (unless one of the interrupts is preemptable). Even if the operations use the same shared variable without protection, a data race cannot occur.
- *Data race between an interrupt and another task*:

  - An operation in an interrupt cannot interfere with an atomic operation in any other task. Even if the operations use the same shared variable without protection, a data race cannot occur.
  - An operation in an interrupt can interfere with a nonatomic operation in any other task unless the other task is also a nonpreemptable interrupt. Therefore, if the operations use the same shared variable without protection, a data race can occur.

For more information, see:

- “Define Task Priorities for Data Race Detection in Polyspace”
- “Concurrency Defects”

A Code Prover verification uses your specifications to determine:

- Whether a global variable is shared.

  See “Global Variables” (Polyspace Code Prover).
- Whether a run-time error can occur.

  For instance, if the operation `var=INT_MAX;` occurs in an interrupt and `var++` occurs in the body of a task, an overflow can occur if the interrupt executes before the operation in the task. The analysis detects the possible overflow.

## Settings

**No Default**

Enter function names or choose from a list.

- Click ✚ to add a field and enter the function name.
- Click 🔍 to list functions in your code. Choose functions from the list.

## Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

## Tips

- In Code Prover, the functions representing interrupts must have the form

  ```
  void functionName (void)
  ```

- If a function `func` takes arguments or returns a value, you cannot use it directly as an interrupt. To use `func` as an interrupt, call `func` from a wrapper `void-void` function and specify the wrapper as an interrupt. See “Configuring Polyspace Multitasking Analysis Manually”.
- If you specify a function as an interrupt, you must provide its definition. Otherwise, a Code Prover verification stops with the error message:

  ```
  task func_name must be a userdef function without parameters
  ```

  A Bug Finder analysis continues but does not consider the function as an interrupt.
- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.
- The Polyspace multitasking analysis assumes that an interrupt cannot interrupt itself.
- Code Prover interprets this option with some limitations. The reason is that Code Prover considers all operations as potentially non-atomic and interruptible. This overapproximation leads to situations where the option might appear to be ignored. For an example, see “Effect of Task Priorities in Code Prover” (Polyspace Code Prover).

### Command-Line Information

**Parameter:** `-interrupts`
**No Default**
**Value:** `*function1*[,*function2*[,...]]`
**Example (Bug Finder):** `polyspace-bug-finder -sources *file_name* -interrupts func_1,func_2`
**Example (Code Prover):** `polyspace-code-prover -sources *file_name* -interrupts func_1,func_2`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources *file_name* -interrupts func_1,func_2`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources *file_name* -interrupts func_1,func_2`

## Version History

**Introduced in R2016b**

### See Also

`-preemptable-interrupts` | `-non-preemptable-tasks` | `Tasks (-entry-points)` | `Cyclic tasks (-cyclic-tasks)` | `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)` | `Show global variable sharing and usage only (-shared-variables-mode)`

#### Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”
“Define Task Priorities for Data Race Detection in Polyspace”

# Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)

Specify routines that disable and reenable interrupts.

## Description

*This option affects a Bug Finder analysis only. The option is not available for code generated from MATLAB code or Simulink models.*

Specify a routine that disables all tasks and interrupts other than the current one, and a routine that reenables them.

The routine that you specify for the option disables preemption by all:

- Non-cyclic tasks.

  See `Tasks (-entry-points)`.
- Cyclic tasks.

  See `Cyclic tasks (-cyclic-tasks)`.
- Interrupts.

  See `Interrupts (-interrupts)`.

In other words, the analysis considers that the body of operations between the disabling routine and the enabling routine is atomic and not interruptible at all.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 2-130 for other options you must also enable.

**Command line and options file**: Use the option `-routine-disable-interrupts` and `-routine-enable-interrupts`. See “Command-Line Information” on page 2-131.

### Why Use This Option

A Bug Finder analysis uses the information when looking for data race defects. For instance, in the following code, the function `disable_all_interrupts` disables all interrupts until the function `enable_all_interrupts` is called. Even if `task`, `isr1` and `isr2` run concurrently, the operations `x=0` or `x=1` cannot interrupt the operation `x++`. There are no data race defects.

```
int x;

void isr1() {
   x = 0;
}

void isr2() {
   x = 1;
}

void task() {
   disable_all_interrupts();
   x++;
   enable_all_interrupts();
}
```

## Settings

**No Default**

- In **Disabling routine**, enter the routine that disables all interrupts.
- In **Enabling routine**, enter the routine that reenables all interrupts.

Enter function names or choose from a list.

- Click ➕ to add a field and enter the function name.
- Click 🔍 to list functions in your code. Choose functions from the list.

## Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

## Tips

- Protection via disabling interrupts is conceptually different from protection via critical sections.

  In the Polyspace multitasking model, to protect two sections of code *from each other* via critical sections, you have to embed them in the same critical section. In other words, you have to place the two sections between calls to the same lock and unlock function.

  For instance, suppose you use critical sections as follows:

  ```
  void isr1() {
     begin_critical_section();
     x = 0;
     end_critical_section();
  }

  void isr2() {
     x = 1;
  }

  void task() {
     begin_critical_section();
     x++;
     end_critical_section();
  }
  ```

Here, the operation x++ is protected from the operation x=0 in isr1, but not from the operation x=1 in isr2. If the function begin_critical_section disabled *all interrupts*, calling it before x++ would have been sufficient to protect it.

Typically, you use one pair of routines in your code to disable and reenable interrupts, but you can have many pairs of lock and unlock functions that implement critical sections.

- The routines that disable and enable interrupts must be functions. For instance, if you define a function-like macro:

  ```
  #define disable_interrupt() interrupt_flag=0
  ```

  You cannot use the macro disable_interrupt() as routine disabling interrupts.
- You can use this option to make sections of nonpreemptable interrupts preemptable. To make only a section of an interrupt preemptable, call a routine enabling all interrupts before that section and call another routine disabling all interrupts after the section is complete.

  For instance, if you specify the routine isr() as an interrupt, it is nonpreemptable by default. However, within isr(), if you call a routine enabling all interrupts, the section following the call is preemptable till you call another routine disabling all interrupts:

  ```
  void isr() {
     x++; //Nonpreemptable
     enable_all_interrupts(); //Routine enabling interrupts
     y++; //Preemptable
     disable_all_interrupts(); //Routine disabling interrupts
     z++; //Nonpreemptable
  }
  ```

## Command-Line Information

**Parameter:** -routine-disable-interrupts | -routine-enable-interrupts
**No Default**
**Value:** *function_name*
**Example (Bug Finder):** polyspace-bug-finder -sources *file_name* -routine-disable-interrupts atomic_section_begins -routine-enable-interrupts atomic_section_ends
**Example (Bug Finder Server):** polyspace-bug-finder-server -sources *file_name* -routine-disable-interrupts atomic_section_begins -routine-enable-interrupts atomic_section_ends

# Version History

**Introduced in R2017a**

## See Also

Tasks (-entry-points) | Cyclic tasks (-cyclic-tasks) | Interrupts (-interrupts) | Critical section details (-critical-section-begin -critical-section-end) | Temporally exclusive tasks (-temporal-exclusions-file) | -non-preemptable-tasks | -preemptable-interrupts

### Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”

“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”
“Define Atomic Operations in Multitasking Code”
“Concurrency Defects”

# Critical section details (-critical-section-begin -critical-section-end)

Specify functions that begin and end critical sections

## Description

*This option is not available for code generated from MATLAB code or Simulink models.*

When verifying multitasking code, Polyspace considers that a critical section lies between calls to a lock function and an unlock function.

```
lock();
/* Critical section code */
unlock();
```

Specify the lock and unlock function names for your critical sections (for instance, `lock()` and `unlock()` in above example).

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 2-134 for other options you must also enable.

**Command line and options file**: Use the option `-critical-section-begin` and `-critical-section-end`. See “Command-Line Information” on page 2-135.

### Why Use This Option

When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait till `my_task` calls the corresponding unlock function. Therefore, critical section operations in the other tasks cannot interrupt critical section operations in `my_task`.

For instance, the operation `var++` in `my_task1` and `my_task2` cannot interrupt each other.

```
int var;

void my_task1() {
   my_lock();
   var++;
   my_unlock();
}

void my_task2() {
   my_lock();
   var++;
   my_unlock();
}
```

Using your specifications, a Code Prover verification checks if your placement of lock and unlock functions protects all shared variables from concurrent access. When determining values of those variables, the verification accounts for the fact that critical sections in different tasks do not interrupt each other.

A Bug Finder analysis uses the critical section information to look for concurrency defects such as data race and deadlock.

## Settings

**No Default**

Click ✚ to add a field.

- In **Starting routine**, enter name of lock function.
- In **Ending routine**, enter name of unlock function.

Enter function names or choose from a list.

- Click ✚ to add a field and enter the function name.
- Click 🔍 to list functions in your code. Choose functions from the list.

## Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

## Tips

- You can also use primitives such as the POSIX functions `pthread_mutex_lock` and `pthread_mutex_unlock` to begin and end critical sections. For a list of primitives that Polyspace can detect automatically, see “Auto-Detection of Thread Creation and Critical Section in Polyspace”.
- For function calls that begin and end critical sections, Polyspace ignores the function arguments.

  For instance, Polyspace treats the two code sections below as the same critical section.

| **Starting routine**: `my_lock` | |
|---|---|
| **Ending routine**: `my_unlock` | |
| `void my_task1() {`<br>`   my_lock(1);`<br>`   /* Critical section code */`<br>`   my_unlock(1);`<br>`}` | `void my_task2() {`<br>`   my_lock(2);`<br>`   /* Critical section code */`<br>`   my_unlock(2);`<br>`}` |

  To work around the limitation, see “Define Critical Sections with Functions That Take Arguments”.
- The functions that begin and end critical sections must be functions. For instance, if you define a function-like macro:

  ```
  #define init() num_locks++
  ```

  You cannot use the macro `init()` to begin or end a critical section.
- When you use multiple critical sections, you can run into issues such as:

  - Deadlock: A sequence of calls to lock functions causes two tasks to block each other.
  - Double lock: A lock function is called twice in a task without an intermediate call to an unlock function.

  Use Polyspace Bug Finder to detect such issues. See "Concurrency Defects".

  Then, use Polyspace Code Prover™ to detect if your placement of lock and unlock functions actually protects all shared variables from concurrent access. See "Global Variables" (Polyspace Code Prover).

- When considering possible values of shared variables, a Code Prover verification takes into account your specifications for critical sections.

  However, if the shared variable is a pointer or array, the software uses the specifications only to determine if the variable is a shared protected global variable. For run-time error checking, the software does not take your specifications into account and considers that the variable can be concurrently accessed.

## Command-Line Information

**Parameter:** `-critical-section-begin` | `-critical-section-end`
**No Default**
**Value:** `*function1*:cs1[,*function2*:cs2[,...]]`
**Example (Bug Finder):** `polyspace-bug_finder -sources *file_name* -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`
**Example (Code Prover):** `polyspace-code-prover -sources *file_name* -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`
**Example (Bug Finder Server):** `polyspace-bug_finder-server -sources *file_name* -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources *file_name* -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`

## See Also

`Tasks (-entry-points)` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Temporally exclusive tasks (-temporal-exclusions-file)` | `-non-preemptable-tasks` | `-preemptable-interrupts`

### Topics

"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Configuring Polyspace Multitasking Analysis Manually"
"Protections for Shared Variables in Multitasking Code"
"Define Atomic Operations in Multitasking Code"
"Define Critical Sections with Functions That Take Arguments"
"Concurrency Defects"
"Global Variables" (Polyspace Code Prover)

# Temporally exclusive tasks (`-temporal-exclusions-file`)

Specify entry point functions that cannot execute concurrently

## Description

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify entry point functions that cannot execute concurrently. The execution of the functions cannot overlap with each other.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See “Dependencies” on page 2-136 for other options you must also enable.

**Command line and options file**: Use the option `-temporal-exclusions-file`. See “Command-Line Information” on page 2-137.

### Why Use This Option

Use this option to implement temporal exclusion in multitasking code.

A Code Prover verification checks if specifying certain tasks as temporally exclusive protects all shared variables from concurrent access. When determining possible values of those shared variables, the verification accounts for the fact that temporally exclusive tasks do not interrupt each other. See “Global Variables” (Polyspace Code Prover).

A Bug Finder analysis uses the temporal exclusion information to look for concurrency defects such as data race. See “Concurrency Defects”.

## Settings

**No Default**

Click ✚ to add a field. In each field, enter a space-separated list of functions. Polyspace considers that the functions in the list cannot execute concurrently.

Enter the function names manually or choose from a list.

- Click ✚ to add a field and enter the function names.
- Click 🔍 to list functions in your code. Choose functions from the list.

## Dependencies

To enable this option in the user interface of the desktop products:

- Select the option Configure multitasking manually .
- Specify function names for Tasks (-entry-points), Cyclic tasks (-cyclic-tasks) and Interrupts (-interrupts).

You can then specify some of these functions as temporally exclusive tasks. Alternatively, if you specify your multitasking configuration using external files with the option External multitasking configuration, some of the functions from your external files can be specified as temporally exclusive.

The ability to specify temporally exclusive tasks is not supported for automatically detected thread creation routines such as pthread_create. These routines can be invoked at different points in the code to create separate threads. However, the temporal exclusion option does not support specifying two separate invocations of the same routine at different points in the code.

## Tips

When considering possible values of shared variables, a Code Prover verification takes into account your specifications for temporally exclusive tasks.

However, if the shared variable is a pointer or array, the software uses the specifications only to determine if the variable is a shared protected global variable. For run-time error checking in Code Prover, the software does not take your specifications into account and considers that the variable can be concurrently accessed.

## Command-Line Information

For the command-line option, create a temporal exclusions file in the following format:

- On each line, enter one group of temporally excluded tasks.
- Within a line, the tasks are separated by spaces.

To enter comments, begin with #. For an example, see the file *polyspaceroot*\polyspace\examples\cxx\Code_Prover_Example\sources\temporal_exclusions.txt. Here, *polyspaceroot* is the Polyspace installation folder, for example C:\Program Files\Polyspace\R2019a.
**Parameter:** -temporal-exclusions-file
**No Default**
**Value:** Name of temporal exclusions file
**Example (Bug Finder):** polyspace-bug-finder -sources *file_name* -temporal-exclusions-file "C:\exclusions_file.txt"
**Example (Code Prover):** polyspace-code-prover -sources *file_name* -temporal-exclusions-file "C:\exclusions_file.txt"
**Example (Bug Finder Server):** polyspace-bug-finder-server -sources *file_name* -temporal-exclusions-file "C:\exclusions_file.txt"
**Example (Code Prover Server):** polyspace-code-prover-server -sources *file_name* -temporal-exclusions-file "C:\exclusions_file.txt"

## See Also

Tasks (-entry-points) | Cyclic tasks (-cyclic-tasks) | Interrupts (-interrupts) | Critical section details (-critical-section-begin -critical-section-end) | -non-preemptable-tasks | -preemptable-interrupts

**Topics**

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”
“Define Atomic Operations in Multitasking Code”
“Concurrency Defects”
“Global Variables” (Polyspace Code Prover)

# Set checkers by file (`-checkers-selection-file`)

Define a custom set of coding standards checks for your analysis

## Description

Specify the full path of a configuration XML file where you define custom selections of coding standards checkers. In the same XML file, you can specify a custom selection of checkers for each of these coding standards:

- MISRA C:2004
- MISRA C:2012
- MISRA C++
- JSF AV C++
- AUTOSAR C++14 *(Bug Finder only)*
- CERT® C *(Bug Finder only)*
- CERT C++ *(Bug Finder only)*
- ISO/IEC TS 17961 *(Bug Finder only)*
- Polyspace Guidelines *(Bug Finder only)*

You can also define custom rules to match identifiers in your code to text patterns you specify.

If you use a selection XML file generated by using a previous version of Polyspace, previously unimplemented checkers might become implemented. Polyspace warns if the file XML file lists an implemented checker as `'notimplemented'`.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

**Command line and options file**: Use the option `-checkers-selection-file`. See “Command-Line Information” on page 2-141.

When you enable this option, set the coding standards that you select to `from-file` to use the specified configuration file.

### Why Use This Option

Use this option to define a selection of coding standard checkers specific to your organization. The configuration of different coding standards is consolidated in a single XML file that you can reuse across projects to enforce common coding standards.

## Settings

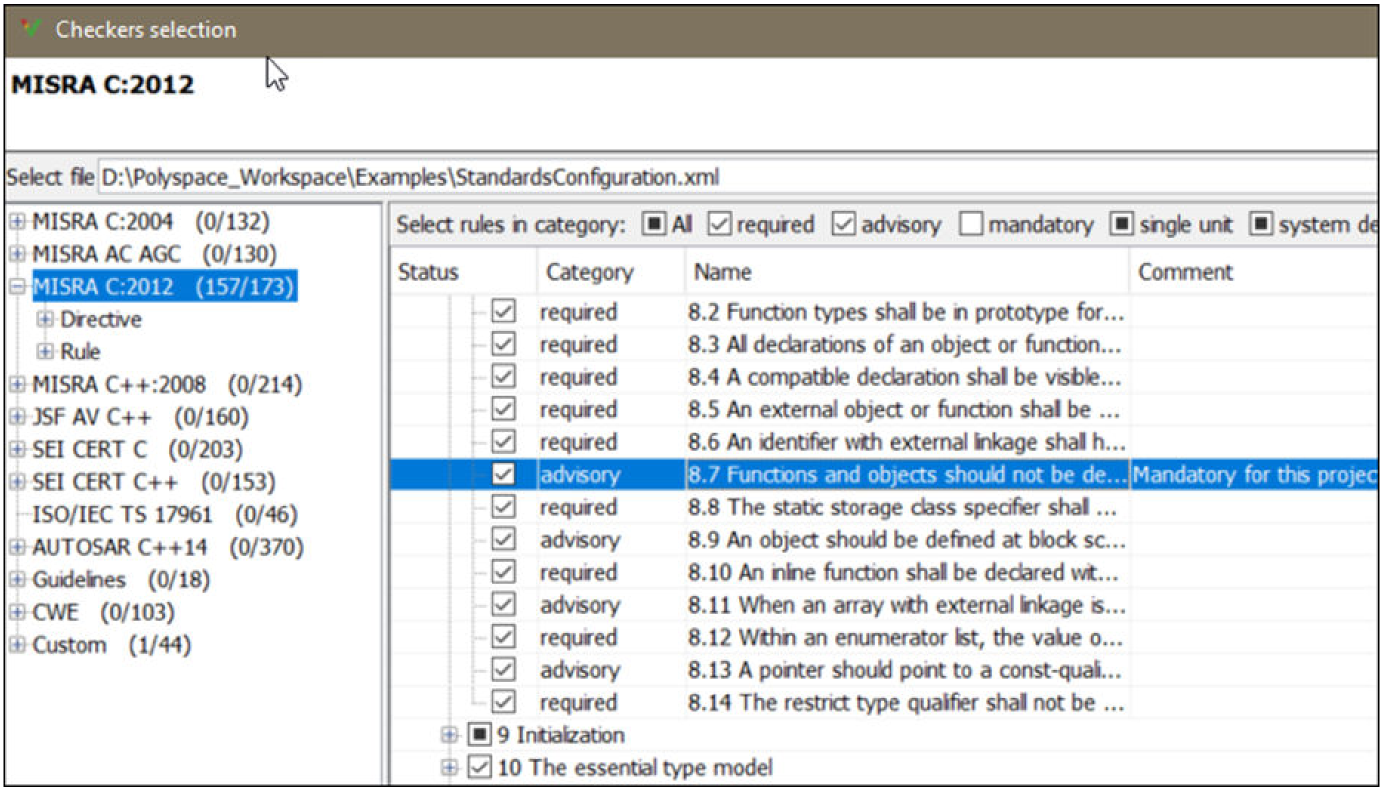
☑ On

Polyspace checks your code against the selection of coding standard checkers, or the custom rules, defined in the configuration file that you specify.

To create a configuration file by using the Polyspace Desktop, in the **Configuration**, select **Coding Standards & Code Metrics**. To open the **Checkers selection** interface, click the folder (📁) on the right pane. Choose the coding standards that you want to configure in the left pane, and then select the rules that you want to activate in the right pane.

To create a configuration file by using Polyspace As you Code IDE plugins, refer to the documentation of your specific plugin.

To use or update an existing file, enter the full path to the file in the in the **Select file** field of the **Checkers selection** dialog box. Alternatively, click **Browse** in the **Checkers selection** window and browse to the existing file.



Checkers selection

MISRA C:2012

Select file D:\Polyspace_Workspace\Examples\StandardsConfiguration.xml

- ⊞ MISRA C:2004 (0/132)
- ⊞ MISRA AC AGC (0/130)
- ⊟ MISRA C:2012 (157/173)
  - ⊞ Directive
  - ⊞ Rule
- ⊞ MISRA C++:2008 (0/214)
- ⊞ JSF AV C++ (0/160)
- ⊞ SEI CERT C (0/203)
- ⊞ SEI CERT C++ (0/153)
- ISO/IEC TS 17961 (0/46)
- ⊞ AUTOSAR C++14 (0/370)
- ⊞ Guidelines (0/18)
- ⊞ CWE (0/103)
- ⊞ Custom (1/44)

Select rules in category: ■ All ☑ required ☑ advisory ☐ mandatory ■ single unit ■ system de

| Status | Category | Name | Comment |
|---|---|---|---|
| ☑ | required | 8.2 Function types shall be in prototype for... | |
| ☑ | required | 8.3 All declarations of an object or function... | |
| ☑ | required | 8.4 A compatible declaration shall be visible... | |
| ☑ | required | 8.5 An external object or function shall be ... | |
| ☑ | required | 8.6 An identifier with external linkage shall h... | |
| ☑ | advisory | 8.7 Functions and objects should not be de... | Mandatory for this projec |
| ☑ | required | 8.8 The static storage class specifier shall ... | |
| ☑ | advisory | 8.9 An object should be defined at block sc... | |
| ☑ | required | 8.10 An inline function shall be declared wit... | |
| ☑ | advisory | 8.11 When an array with external linkage is... | |
| ☑ | required | 8.12 Within an enumerator list, the value o... | |
| ☑ | advisory | 8.13 A pointer should point to a const-quali... | |
| ☑ | required | 8.14 The restrict type qualifier shall not be ... | |

⊞ ■ 9 Initialization

⊞ ☑ 10 The essential type model

☐ Off (default)

Polyspace does not check your code against the selection of coding standard checkers, or the custom rules, defined in the configuration file you specify.

## Tips

- For the Polyspace desktop products, specify the coding standard configuration in the Polyspace User Interface. When you save the configuration, an XML file is created for use in the current and other projects.
- To create a custom coding standard classification, or to tag coding rule checkers of interest, enter text in the **Comment** column of the **Checkers selection** window. Polyspace displays that text in the **Results Details** pane and in the **Detail** column of the **Results List** (if available) when you review results in the desktop interface, in Polyspace Access, or in the Polyspace as You Code plugins.

- For the Polyspace Server products, you have to create a coding standard XML. Depending on the standard that you want to enable, make a writeable copy of one of the files in *polyspaceserverroot*\help\toolbox\bugfinder\examples\coding_standards_XML. Enable specific rules by using entries in the XML file (all rules from a standard are disabled in the template). Here, *polyspaceserverroot* is the root installation folder for the Polyspace Server products, for instance, C:\Program Files\Polyspace Server\R2023a.

  For instance, to turn off MISRA C:2012 rule 8.1, in the file misra_c_2012_rules.xml, use this entry:

  ```
  <standard name="MISRA C:2012">
    ...
    <section name="8 Declarations and definitions">
        ...
        <check id="8.1" state="off">
        </check>
        ...
    </section>
    ...
  </standard>
  ```

  For a full list of rule IDs and section names, see:

  - “AUTOSAR C++14 Rules”
  - “CERT C Rules and Recommendations”
  - “CERT C++ Rules”
  - “ISO/IEC TS 17961 Rules”
  - “Custom Coding Rules”
  - “JSF C++ Rules”
  - “MISRA C:2004 Rules”
  - “MISRA C:2012 Directives and Rules”
  - “MISRA C++:2008 Rules”
  - “Guidelines”

**Note** The XML format of the checker configuration file might change in future releases.

## Command-Line Information

Use the command -checkers-selection-file in the command line to select a custom set of coding standards checks for your analysis.
**Parameter:** -checkers-selection-file
**Value:** Full path of XML configuration file
**Default:** Off
**Example (Bug Finder):** polyspace-bug-finder -sources *file_name* -checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file
**Example (Bug Finder Server):** polyspace-bug-finder-server -sources *file_name* -checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file

Alternatively, activate the custom selection of coding rules and Bug Finder defects in an activation XML file. See -checkers-activation-file.

## See Also

`Do not generate results for (-do-not-generate-results-for)` | `-checkers-activation-file`

### Topics

"Specify Polyspace Analysis Options"
"Check for and Review Coding Standard Violations"

# Check MISRA C:2004 (`-misra2`)

Check for violation of MISRA C:2004 rules

## Description

Specify whether to check for violation of MISRA C:2004 rules[1]. Each value of the option corresponds to a subset of rules to check.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-144 for other options that you must also enable.

**Command line and options file**: Use the option `-misra2`. See “Command-Line Information” on page 2-144.

### Why Use This Option

Use this option to specify the subset of MISRA C:2004 rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

## Settings

**Default:** `required-rules`

`required-rules`

Check required coding rules.

`single-unit-rules`

Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

`system-decidable-rules`

Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

`all-rules`

Check required and advisory coding rules.

`SQO-subset1`

Check only a subset of MISRA C rules. For more information, see “Software Quality Objective Subsets (C:2004)”.

[1] MISRA and MISRA C are registered trademarks of MIRA Ltd., held on behalf of the MISRA Consortium.

`SQO-subset2`

Check a subset of rules including `SQO-subset1` and some additional rules. For more information, see “Software Quality Objective Subsets (C:2004)”.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click Edit, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Checkers selection** window. Save the file.

To use or update an existing configuration file, in the **Checkers selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

## Dependencies

- This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

  For projects with mixed C and C++ code, the MISRA C:2004 checker analyzes only `.c` files.
- If you set `Source code language (-lang)` to `C-CPP`, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

## Tips

If you select the option `single-unit-rules` or `system-decidable-rules` and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see “Coding Rule Subsets Checked Early in Analysis”.

## Command-Line Information

Use the command `-misra2` in the command line to check for violations of MISRA C:2004 rules.
**Parameter:** `-misra2`
**Value:** `required-rules` | `all-rules` | `SQO-subset1` | `SQO-subset2` | `single-unit-rules` | `system-decidable-rules` | `from-file`
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-misra2 all-rules`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-misra2 all-rules`

Alternatively, enable all or specific MISRA C:2004 rules through a checkers activation XML file. See `-checkers-activation-file`.

# Version History

**R2021b: Use of text format for coding rules file not supported**
*Errors starting in R2021b*

Since R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. Using a text file for this purpose results in an error. You can save custom selections for all the coding standards that Polyspace supports in the same file.

**Desktop interface:**

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click 📁. In the **Checkers selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *`filename`*`.xml`, where *`filename`* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

**Command-line/ IDEs:**

In the command-line or in the IDE extensions, using text files as input to `-misra2` results in an error. To select a custom selection of MISRA C:2004 rules, use an XML file.

Use the file `misra_c_2004_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *`polyspaceroot`*`\help\toolbox\bugfinder\examples\coding_standards_XML` . Here, *`polyspaceroot`* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2023a`. To update your script, see this table

| Option | Use Instead |
|---|---|
| `-misra2 "custom_standard.conf"` | `-checkers-selection-file misra_c_2004_rules.xml -misra2 from-file` |

See:

- “Configure Coding Rules Checking”
- “Setting Checkers in Polyspace as You Code”

**Note** The XML format of the checker configuration file can change in future releases.

### Example of Configuration File in XML Format

To turn on MISRA C:2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For a full list of rule IDs and section names, see:

- “AUTOSAR C++14 Rules”
- “CERT C Rules and Recommendations”
- “CERT C++ Rules”
- “ISO/IEC TS 17961 Rules”
- “Custom Coding Rules”
- “JSF C++ Rules”
- “MISRA C:2004 Rules”
- “MISRA C:2012 Directives and Rules”
- “MISRA C++:2008 Rules”
- “Guidelines”

### R2019b: Use of text format for coding rules file will not be supported

*Warns starting in R2019b*

Since R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file. If you use a text file, Polyspace issues a warning and converts the text file to XML.

## See Also

`Do not generate results for (-do-not-generate-results-for)`

### Topics

“Specify Polyspace Analysis Options”
“Check for and Review Coding Standard Violations”
“MISRA C:2004 Rules”

# Check MISRA AC AGC (-misra-ac-agc)

Check for violation of MISRA AC AGC rules

## Description

Specify whether to check for violation of rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*. Each value of the option corresponds to a subset of rules to check.

To check for MISRA C:2012 rules that apply to generated code, use the option Use generated code requirements (-misra3-agc-mode).

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-148 for other options that you must also enable.

**Command line and options file**: Use the option -misra-ac-agc. See “Command-Line Information” on page 2-148.

### Why Use This Option

Use this option to specify the subset of MISRA C:2004 AC AGC rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

## Settings

**Default**: OBL-rules

OBL-rules

Check required coding rules.

OBL-REC-rules

Check required and recommended rules.

single-unit-rules

Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

system-decidable-rules

Check rules in the single-unit-rules subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

all-rules

Check required, recommended and readability-related rules.

`SQO-subset1`

Check a subset of rules. For more information, see “Software Quality Objective Subsets (AC AGC)”.

`SQO-subset2`

Check a subset of rules including `SQO-subset1` and some additional rules. For more information, see “Software Quality Objective Subsets (AC AGC)”.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click Edit, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Checkers selection** window. Save the file.

To use or update an existing configuration file, in the **Checkers selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

## Dependencies

- This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

  For projects with mixed C and C++ code, the MISRA AC AGC checker analyzes only `.c` files.
- If you set `Source code language (-lang)` to `C-CPP`, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

## Tips

If you select the option `single-unit-rules` or `system-decidable-rules` and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see “Coding Rule Subsets Checked Early in Analysis”.

## Command-Line Information

Use the command `-misra-ac-agc` in the command line to check for violations of MISRA AC AGC rules.
**Parameter:** `-misra-ac-agc`
**Value:** `OBL-rules` | `OBL-REC-rules` | `single-unit-rules` | `system-decidable-rules` | `all-rules` | `SQO-subset1` | `SQO-subset2` | `from-file`
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-misra-ac-agc all-rules`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-misra-ac-agc all-rules`

Alternatively, activate the rules in an activation XML file. See `-checkers-activation-file`.

## Version History

**R2021b: Using text format for coding rules file not supported**
*Errors starting in R2021b*

Since R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. Use of a text file for this porpose results in an error. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace no longer supports custom coding standard files in text format.

**Desktop interface:**

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click 📁. In the **Checkers selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as `*filename*.xml`, where `*filename*` is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

**Command-line/ IDEs:**

In the command-line or in the IDE extensions, using text files as input to `-misra-ac-agc` results in an error. To select a custom selection of MISRA AC AGC rules, use an XML file.

Use the file `misra_ac_agc_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in `*polyspaceroot*\help\toolbox\bugfinder\examples\coding_standards_XML`. Here, *`polyspaceroot`* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2023a`. To update your script, see this table

| Option | Use Instead |
|---|---|
| `-misra-ac-agc "custom_standard.conf"` | `-checkers-selection-file misra_ac_agc_rules.xml -misra-ac-agc from-file` |

See:

- "Configure Coding Rules Checking"
- "Setting Checkers in Polyspace as You Code"

**Note** The XML format of the checker configuration file can change in future releases.

### Example of Configuration File in XML Format

To turn on MISRA C:2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For a full list of rule IDs and section names, see:

- “AUTOSAR C++14 Rules”
- “CERT C Rules and Recommendations”
- “CERT C++ Rules”
- “ISO/IEC TS 17961 Rules”
- “Custom Coding Rules”
- “JSF C++ Rules”
- “MISRA C:2004 Rules”
- “MISRA C:2012 Directives and Rules”
- “MISRA C++:2008 Rules”
- “Guidelines”

### R2019b: Use of text format for coding rules file will not be supported

*Warns starting in R2019b*

Since R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file. If you use a text file, Polyspace issues a warning and converts the text file to XML.

## See Also

`Do not generate results for (-do-not-generate-results-for)`

### Topics

“Specify Polyspace Analysis Options”
“Check for and Review Coding Standard Violations”
“MISRA C:2004 Rules”

# Check MISRA C:2012 (-misra3)

Check for violations of MISRA C:2012 rules and directives

## Description

Specify whether to check for violations of MISRA C:2012 guidelines[2]Each value of the option corresponds to a subset of guidelines to check.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-152 for other options that you must also enable.

**Command line and options file**: Use the option `-misra3`. See “Command-Line Information” on page 2-153.

### Why Use This Option

Use this option to specify the subset of MISRA C:2012 rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

## Settings

**Default:** `mandatory-required`

`mandatory`

Check for mandatory guidelines.

`mandatory-required`

Check for mandatory and required guidelines.

- Mandatory guidelines: Your code must comply with these guidelines.
- Required guidelines: You may deviate from these guidelines. However, you must complete a formal deviation record, and your deviation must be authorized.

  See Section 5.4 of the MISRA C:2012 guidelines. For an example of a deviation record, see Appendix I of the MISRA C:2012 guidelines.

**Note** To turn off some required guidelines, instead of `mandatory-required` select `custom`. To clear specific guidelines, click Edit. In the **Comment** column, enter your rationale for disabling a guideline. For instance, you can enter the Deviation ID that refers to a deviation record for the guideline. The rationale appears in your generated report.

[2] MISRA and MISRA C are registered trademarks of MIRA Ltd., held on behalf of the MISRA Consortium.

`single-unit-rules`

Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

`system-decidable-rules`

Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

`all`

Check for mandatory, required, and advisory guidelines.

`SQO-subset1`

Check for only a subset of guidelines. For more information, see “Software Quality Objective Subsets (C:2012)”.

`SQO-subset2`

Check for the subset `SQO-subset1`, plus some additional rules. For more information, see “Software Quality Objective Subsets (C:2012)”.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click Edit, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Checkers selection** window. Save the file.

To use or update an existing configuration file, in the **Checkers selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

## Dependencies

- This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

  For projects with mixed C and C++ code, the MISRA C:2012 checker analyzes only `.c` files.
- If you set `Source code language (-lang)` to `C-CPP`, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

## Tips

- If you select the option `single-unit-rules` or `system-decidable-rules` and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see “Coding Rule Subsets Checked Early in Analysis”.
- In code generated by using Embedded Coder®, there are known deviations from MISRA C:2012. See “Deviations Rationale for MISRA C:2012 Compliance” (Embedded Coder).

## Command-Line Information

Use the command `-misra3` in the command line to check for violations of MISRA C:2012 rules.
**Parameter:** `-misra3`
**Value:** `mandatory` | `mandatory-required` | `single-unit-rules` | `system-decidable-rules` | `all` | `SQO-subset1` | `SQO-subset2` | `from-file`
**Example (Bug Finder):** `polyspace-bug-finder -lang c -sources` *`file_name`* `-misra3 mandatory-required`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -lang c -sources` *`file_name`* `-misra3 mandatory-required`

Alternatively, enable all or specific MISRA C:2012 rules through a checkers activation XML file. See `-checkers-activation-file`.

# Version History

### R2021b: Using text format for coding rules file not supported

*Errors starting in R2021b*

Since R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file. Using a text formal file for this purpose results in an error.

**Desktop interface:**

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click 📁. In the **Checkers selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *`filename`*`.xml`, where *`filename`* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

**Command-line/ IDEs:**

In the command-line or in the IDE extensions, using text files as input to `-misra3` results in an error. To select a custom selection of MISRA C:2012 rules and directives, use an XML file.

Use the file `misra_c_2012_rules.xml` as a template to create the XML file where you define the custom selection. This template file is located in *`polyspaceroot`*`\help\toolbox\bugfinder\examples\coding_standards_XML`. Here, *`polyspaceroot`* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2023a`. To update your script, see this table

| Option | Use Instead |
|---|---|
| `-misra3 "custom_standard.conf"` | `-checkers-selection-file misra_c_2012_rules.xml -misra3 from-file` |

See:

- "Configure Coding Rules Checking"
- "Setting Checkers in Polyspace as You Code"

**Note** The XML format of the checker configuration file can change in future releases.

### Example of Configuration File in XML Format

To turn on MISRA C:2012 rule 8.1, use this entry in the XML file:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For a full list of rule IDs and section names, see:

- "AUTOSAR C++14 Rules"
- "CERT C Rules and Recommendations"
- "CERT C++ Rules"
- "ISO/IEC TS 17961 Rules"
- "Custom Coding Rules"
- "JSF C++ Rules"
- "MISRA C:2004 Rules"
- "MISRA C:2012 Directives and Rules"
- "MISRA C++:2008 Rules"
- "Guidelines"

### R2019b: Use of text format for coding rules file will not be supported

*Warns starting in R2019b*

Since R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file. If you use a text file, Polyspace issues a warning and converts the text file to XML.

## See Also

`Do not generate results for (-do-not-generate-results-for)`

### Topics

“Specify Polyspace Analysis Options”
“Check for and Review Coding Standard Violations”
“MISRA C:2012 Directives and Rules”

# Use generated code requirements (-misra3-agc-mode)

Check for violations of MISRA C:2012 rules and directives that apply to generated code

## Description

Specify whether to use the MISRA C:2012 categories for automatically generated code. This option changes which rules are mandatory, required, or advisory.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See "Dependency" on page 2-157 for other options that you must also enable.

**Command line and options file**: Use the option `-misra3-agc-mode`. See "Command-Line Information" on page 2-157.

### Why Use This Option

Use this option to specify that you are checking for MISRA C:2012 rules in generated code. The option modifies the MISRA C:2012 subsets so that they are tailored for generated code.

## Settings

□ Off (default)

Use the normal categories (mandatory, required, advisory) for MISRA C:2012 coding guideline checking.

☑ On (default for analyses from Simulink)

Use the generated code categories (mandatory, required, advisory, readability) for MISRA C:2012 coding guideline checking.

For analyses started from the Simulink plug-in, this option is the default value.

### Category changed to `Advisory`

These rules are changed to advisory:

- 5.3
- 7.1
- 8.4, 8.5, 8.14
- 10.1, 10.2, 10.3, 10.4, 10.6, 10.7, 10.8
- 14.1, 14.4
- 15.2, 15.3
- 16.1, 16.2, 16.3, 16.4, 16.5, 16.6, 16.7

- 20.8

**Category changed to Readability**

These guidelines are changed to readability:

- Dir 4.5
- 2.3, 2.4, 2.5, 2.6, 2.7
- 5.9
- 7.2, 7.3
- 9.2, 9.3, 9.5
- 11.9
- 13.3
- 14.2
- 15.7
- 17.5, 17.7, 17.8
- 18.5
- 20.5

## Dependency

To use this option, activate at least one MISRA C:2012 rule. To activate MISRA C:2012 rules, use either of these:

- Use the option `Check MISRA C:2012 (-misra3)` to activate a preselected subset of the rules.
- Use the option `Set checkers by file (-checkers-selection-file)` alongside `Check MISRA C:2012 (-misra3)` to activate a custom selection that is specified in an XML file.

When using an XML file to specify a custom selection, select at least one MISRA C:2012 rule in the file.

## Command-Line Information

Use the command `-misra3-agc-mode` in the command line to check for violations of MISRA C:2012 rules and directives that apply to generated code.
**Parameter:** `-misra3-agc-mode`
**Default:** Off
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-misra3 all -misra3-agc-mode`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-misra3 all -misra3-agc-mode`

Alternatively, activate the rules and directives in an activation XML file. See `-checkers-activation-file`.

## See Also

`Do not generate results for (-do-not-generate-results-for)` | `Check MISRA C:2012 (-misra3)`

### Topics

"Specify Polyspace Analysis Options"
"Check for and Review Coding Standard Violations"
"MISRA C:2012 Directives and Rules"

# Effective boolean types (-boolean-types)

Specify data types that coding rule checker must treat as effectively Boolean

## Description

Specify data types that the coding rule checker must treat as effectively Boolean. You can specify a data type as effectively Boolean only if you have defined it through an `enum` or `typedef` statement in your source code.

Polyspace natively supports these boolean types, depending on your language:

- Type `_Bool` in C99 once you include `stdbool.h` in one of your source files.
- Type `bool` in C11 and for all versions of C++.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-160 for other options that you must also enable.

**Command line and options file**: Use the option `-boolean-types`. See “Command-Line Information” on page 2-160.

### Why Use This Option

Use this option to allow Polyspace to check the following coding rules:

- MISRA C: 2004 and MISRA AC AGC

| Rule Number | Rule Statement |
|---|---|
| 12.6 | Operands of logical operators, &&, `||`, and `!`, should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to other operators. |
| 13.2 | Tests of a value against zero should be made explicit, unless the operand is effectively Boolean. |
| 15.4 | A `switch` expression should not represent a value that is effectively Boolean. |

- MISRA C: 2012

| Rule Number | Rule Statement |
|---|---|
| 10.1 | Operands shall not be of an inappropriate essential type |
| 10.3 | The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category |
| 10.5 | The value of an expression should not be cast to an inappropriate essential type |

| Rule Number | Rule Statement |
|---|---|
| 14.4 | The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type. |
| 16.7 | A switch-expression shall not have essentially Boolean type. |

For example, in the following code, Polyspace detects a violation of MISRA C: 2012 rule 14.4 because `boolean_T` is not recognized as effective boolean. If you rerun the analysis and specify option `-boolean-types boolean_T`, Polyspace considers that the code is compliant with rule 14.4.

```
typedef int boolean_T;

void func1(void);
void func2(void);

void func(boolean_T flag) {
    if(flag) // No misra2012 14.4 violation when you use -boolean-types
        func1();
    else
        func2();
}
```

## Settings

**No Default**

Click + to add a field. Enter a type name that you want Polyspace to treat as Boolean.

## Dependencies

This option is enabled only if you select one of these options:

- `Check MISRA C:2004 (-misra2)`
- `Check MISRA AC AGC (-misra-ac-agc)`.
- `Check MISRA C:2012 (-misra3)`

## Command-Line Information

**Parameter:** `-boolean-types`
**Value:** *type1*[,*type2*[,...]]
**No Default**
**Example (Bug Finder):** `polyspace-bug-finder -sources` *filename* `-misra2 required-rules -boolean-types boolean1_t,boolean2_t`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *filename* `-misra2 required-rules -boolean-types boolean1_t,boolean2_t`

## See Also

`Check MISRA C:2004 (-misra2)` | `Check MISRA AC AGC (-misra-ac-agc)` | `Check MISRA C:2012 (-misra3)`

### Topics

“Specify Polyspace Analysis Options”
“Check for and Review Coding Standard Violations”
“MISRA C:2004 Rules”
“MISRA C:2012 Directives and Rules”

# Allowed pragmas (`-allowed-pragmas`)

Specify pragma directives that are documented

## Description

Specify pragma directives that must not be flagged by MISRA C:2004 rule 3.4 or MISRA C++ rule 16-6-1. These rules require that you document all pragma directives.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-162 for other options that you must also enable.

**Command line and options file**: Use the option `-allowed-pragmas`. See “Command-Line Information” on page 2-163.

### Why Use This Option

MISRA C:2004/MISRA AC AGC rule 3.4 and MISRA C++ rule 16-6-1 require that all pragma directives are documented within the documentation of the compiler. If you list a pragma as documented using this analysis option, Polyspace does not flag use of the pragma as a violation of these rules.

## Settings

**No Default**

Click + to add a field. Enter the pragma name that you want Polyspace to ignore during coding rule checking .

## Dependencies

This option is enabled only if you select one of these options:

- `Check MISRA C:2004 (-misra2)`
- `Check MISRA AC AGC (-misra-ac-agc)`.
- `Check MISRA C++:2008 (-misra-cpp)`

## Tips

Enter the name of the pragma only excluding any argument. For instance, if you use the pragma `pack`:

```
#pragma pack(n)
```

Enter only the name `pack` for this option.

## Command-Line Information

**Parameter:** `-allowed-pragmas`
**Value:** `pragma1[,pragma2[,...]]`
**Example (Bug Finder):** `polyspace-bug-finder -sources filename -misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources filename -misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`

## See Also

Check MISRA C:2004 (-misra2) | Check MISRA AC AGC (-misra-ac-agc) | Check MISRA C++:2008 (-misra-cpp)

### Topics

"Specify Polyspace Analysis Options"
"Check for and Review Coding Standard Violations"
"MISRA C:2004 Rules"
"MISRA C++:2008 Rules"

# Check custom rules (`-custom-rules`)

Follow naming conventions for identifiers

## Description

Define naming conventions for identifiers and check your code against them.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

**Command line and options file**: Use the option `-custom-rules`. See “Command-Line Information” on page 2-166.

### Why Use This Option

Use this option to impose naming conventions on identifiers. Using a naming convention allows you to easily determine the nature of an identifier from its name. For instance, if you define a naming convention for structures, you can easily tell whether an identifier represents a structured variable or not.

After analysis, the **Results List** pane lists violations of the naming conventions. On the **Source** pane, for every violation, Polyspace assigns a ▼ symbol to the keyword or identifier relevant to the violation.

For the full list of types on which you can apply naming conventions, see “Custom Coding Rules”.

## Settings

☑ On

Polyspace matches identifiers in your code against text patterns you define. Define the text patterns in a custom coding rules file. To create a coding rules file,

- Use the custom rules wizard:

  1 Click Edit. A **Checkers selection** window opens.

  2 The **Custom** node in the left pane is highlighted. Expand the nodes in the right pane to select custom rule you want to check.

  3 For every custom rule you want to check:

     a Select **On** ☑.

     b In the **Convention** column, enter the error message you want to display if the rule is violated.

       For example, for rule 4.3, **All struct fields must follow the specified pattern**, you can enter `All struct fields must begin with s_`. This message appears on the **Result Details** pane if:

- You specify the **Pattern** as s_[A-Za-z0-9_]+.
- A structure field in your code does not begin with s_.

c In the **Pattern** column, enter the text pattern.

For example, for rule 4.3, **All struct fields must follow the specified pattern**, you can enter s_[A-Za-z0-9_]+. Polyspace reports violation of rule 4.3 if a structure field does not begin with s_.

You can use Perl regular expressions to define patterns. For instance, you can use the following expressions.

| Expression | Meaning |
|---|---|
| . | Matches any single character except newline |
| [a-z0-9] | Matches any single letter in the set a-z, or digit in the set 0-9 |
| [^a-e] | Matches any single letter not in the set a-e |
| \d | Matches any single digit |
| \w | Matches any single alphanumeric character or _ |
| x? | Matches 0 or 1 occurrence of x |
| x* | Matches 0 or more occurrences of x |
| x+ | Matches 1 or more occurrences of x |

For frequent patterns, you can use the following regular expressions:

- (?!__)[a-z0-9_]+(?!__), matches a text pattern that does not start and end with two underscores.

```
int __text; //Does not match
int _text_; //Matches
```

- [a-z0-9_]+_(u8|u16|u32|s8|s16|s32) , matches a text pattern that ends with a specific suffix.

```
int _text_; //Does not match
int _text_s16; //Matches
int _text_s33; // Does not match
```

- [a-z0-9_]+_(u8|u16|u32|s8|s16|s32)(_b3|_b8)? , matches a text pattern that ends with a specific suffix and an optional second suffix.

```
int _text_s16; //Matches
int _text_s16_b8; //Matches
```

If you do not want Polyspace to report a trivial case when you check for a specific pattern, add the pattern for that trivial case to your regular expression.

For example, for rule 8.5, **All floating constants must follow the specified pattern**, the specified pattern might be that all floating point constants must be suffixed with "f" or "F". To ignore the trivial case 0.0, use this regular expression for instance:

```
(.*(f|F))|(^0\.0$)
```

. Polyspace reports violations of this rule for the specified pattern except for 0.0.

```
float arr[] = { 0.0, // No violation reported
                10.05, // Violation reported
                4.00f }; // No violation reported
```

For a complete list of regular expressions, see Perl documentation.

To use or update an existing coding rules file, click Edit to open the **Checkers selection** window then do one of the following:

- Enter the full path to the file in the field provided
- Click **Browse** and navigate to the file location.

☐ Off (default)

Polyspace does not check your code against custom naming conventions.

## Command-Line Information

Use the command `-custom-rules` in the command line to check for violations of predefined naming conventions.
**Parameter:** `-custom-rules`
**Value:** `from-file`, specify the file using `Set checkers by file (-checkers-selection-file)`
**Default**: Off
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"`

Alternatively, activate the rules in an activation XML file. See `-checkers-activation-file`.

# Version History

**R2021b: Use of text format for coding rules file not supported**
*Errors starting in R2021b*

Since R2019a, the file where you define custom coding rules uses the XML format. Using a text file for this purpose results in an error. You can save selections for custom coding rules and all the coding standards that Polyspace supports in the same file.

**Desktop user interface:**

If you have a project that contains custom coding rules and coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click 📁. In the **Checkers selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename*.xml, where *filename* is the name of the first selected file alphabetically. For instance, if you select foo.conf and bar.conf, they are saved as bar.conf.xml.

**Command-line:**

In the command-line or in the IDE extensions, using text files as input to -custom-rules results in an error. To select a custom selection of custom rules, use an XML file.

Use the file custom_rules.xml as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *polyspaceroot*\help\toolbox\bugfinder\examples\coding_standards_XML. Here, *polyspaceroot* is the root installation folder for the Polyspace products, for instance, C:\Program Files\Polyspace\R2023a. To update your script, replace reference to the old file format with the new XML file format .

### Example of Configuration File in XML Format

To turn on and define custom coding rule 8.1, use this entry:

```
<standard name="CUSTOM RULES">
  ...
  <section name="8 Constants">
      ...
      <check id="8.1" state="on">
       </check>
      ...
  </section>
  ...
</standard>
```

For a full list of rule IDs and section names, see:

- “AUTOSAR C++14 Rules”
- “CERT C Rules and Recommendations”
- “CERT C++ Rules”
- “ISO/IEC TS 17961 Rules”
- “Custom Coding Rules”
- “JSF C++ Rules”
- “MISRA C:2004 Rules”
- “MISRA C:2012 Directives and Rules”
- “MISRA C++:2008 Rules”
- “Guidelines”

### R2019b: Use of text format for coding rules file will not be supported

*Warns starting in R2019b*

Since R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file. If you use a text file, Polyspace issues a warning and converts the text file to XML.

## See Also

**Topics**
“Specify Polyspace Analysis Options”
“Check for and Review Coding Standard Violations”
“Create Custom Coding Rules”
“Custom Coding Rules”

# Check MISRA C++:2008 (`-misra-cpp`)

Check for violations of MISRA C++ rules

## Description

Specify whether to check for violation of MISRA C++:2008 rules [3]. Each value of the option corresponds to a subset of rules to check.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependency” on page 2-170 for other options that you must also enable.

**Command line and options file**: Use the option `-misra-cpp`. See “Command-Line Information” on page 2-170.

### Why Use This Option

Use this option to specify the subset of MISRA C++ rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

## Settings

**Default:** `required-rules`

`required-rules`

Check required coding rules.

`all-rules`

Check required and advisory coding rules.

`SQO-subset1`

Check only a subset of MISRA C++ rules. For more information, see “Software Quality Objective Subsets (C++)”.

`SQO-subset2`

Check a subset of rules including `SQO-subset1` and some additional rules. For more information, see “Software Quality Objective Subsets (C++)”

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click Edit, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Checkers selection** window. Save the file.

[3] MISRA is a registered trademark of MIRA Ltd., held on behalf of the MISRA Consortium.

To use or update an existing configuration file, in the **Checkers selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

### Dependency

This option is available only if you set `Source code language (-lang)` to `CPP` or `C-CPP`.

For projects with mixed C and C++ code, the MISRA C++ checker analyzes only `.cpp` files.

### Command-Line Information

Use the command `-misra-cpp` in the command line to check for violations of MISRA C++ rules.
**Parameter:** `-misra-cpp`
**Value:** `required-rules` | `all-rules` | `SQO-subset1` | `SQO-subset2` | `from-file`
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-misra-cpp all-rules`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-misra-cpp all-rules`

Alternatively, enable all or specific MISRA C++ rules through a checkers activation XML file. See `-checkers-activation-file`.

## Version History

**R2021b: Use of text format for coding rules file not supported**
*Errors starting in R2021b*

Since R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. Use of text format for this purpose results in an error. You can save custom selections for all the coding standards that Polyspace supports in the same file.

**Desktop interface:**

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click 📁. In the **Checkers selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *`filename`*`.xml`, where *`filename`* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

**Command-line:**

In the command-line or in the IDE extensions, using text files as input to `-misra-cpp` results in an error. To select a custom selection of MISRA C++:2008 rules, use an XML file.

Use the file `misra_cpp_2008_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in `*polyspaceroot*\help\toolbox\bugfinder\examples\coding_standards_XML`. Here, `*polyspaceroot*` is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2023a`. To update your script, see this table

| Option | Use Instead |
|---|---|
| `-misra-cpp "custom_standard.conf"` | `-checkers-selection-file misra_cpp_2008_rules.xml -misra-cpp from-file` |

.

**Note** The XML format of the checker configuration file can change in future releases.

### Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For a full list of rule IDs and section names, see:

- “AUTOSAR C++14 Rules”
- “CERT C Rules and Recommendations”
- “CERT C++ Rules”
- “ISO/IEC TS 17961 Rules”
- “Custom Coding Rules”
- “JSF C++ Rules”
- “MISRA C:2004 Rules”
- “MISRA C:2012 Directives and Rules”
- “MISRA C++:2008 Rules”
- “Guidelines”

### R2019b: Use of text format for coding rules file will not be supported

*Warns starting in R2019b*

Since R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file. If you use a text file, Polyspace issues a warning and converts the text file to XML.

## See Also

`Do not generate results for (-do-not-generate-results-for)`

### Topics

"Specify Polyspace Analysis Options"
"Check for and Review Coding Standard Violations"
"MISRA C++:2008 Rules"

# Check JSF AV C++ rules (`-jsf-coding-rules`)

Check for violations of JSF C++ rules

## Description

Specify whether to check for violation of JSF AV C++ rules (JSF++:2005)[4]. Each value of the option corresponds to a subset of rules to check.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependency” on page 2-174 for other options that you must also enable.

**Command line and options file**: Use the option `-jsf-coding-rules`. See “Command-Line Information” on page 2-174.

### Why Use This Option

Use this option to specify the subset of JSF C++ rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

## Settings

**Default:** `shall-rules`

`shall-rules`

Check all **Shall** rules. **Shall** rules are mandatory requirements and require verification.

`shall-will-rules`

Check all **Shall** and **Will** rules. **Will** rules are intended to be mandatory requirements but do not require verification.

`all-rules`

Check all **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click Edit, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Checkers selection** window. Save the file.

To use or update an existing configuration file, in the **Checkers selection** window, enter the full path to the file in the field provided or click **Browse**.

[4] JSF and Joint Strike Fighter® are registered trademarks of Lockheed Martin®.

If you set the option to from-file, enable Set checkers by file (-checkers-selection-file).

## Tips

- If your project uses a setting other than generic for Compiler (-compiler), some rules might not be completely checked. For example, AV Rule 8: “All code shall conform to ISO/IEC 14882:2002(E) standard C++.”

## Dependency

This option is available only if you set Source code language (-lang) to CPP or C-CPP.

For projects with mixed C and C++ code, the JSF C++ checker analyzes only .cpp files.

## Command-Line Information

Use the command -jsf-coding-rules in the command line to check for violations of JSF C++ rules.

**Parameter:** -jsf-coding-rules

**Value:** shall-rules | shall-will-rules | all-rules | from-file

**Example (Bug Finder):** polyspace-bug-finder -sources *file_name* -jsf-coding-rules all-rules

**Example (Bug Finder Server):** polyspace-bug-finder-server -sources *file_name* -jsf-coding-rules all-rules

Alternatively, enable all or specific JSF C++ rules through a checkers activation XML file. See -checkers-activation-file.

# Version History

**R2021b: Using text format for coding rules file will not be supported**

*Errors starting in R2021b*

Since R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. Use of a text file for this purpose results in an error. You can save custom selections for all the coding standards that Polyspace supports in the same file.

**Desktop interface:**

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click 📁. In the **Checkers selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename*.xml, where *filename* is the name of the first selected file alphabetically. For instance, if you select foo.conf and bar.conf, they are saved as bar.conf.xml.

**Command-line/ IDEs:**

In the command-line or in the IDE extensions, using text files as input to `-jsf-coding-rules` results in an error. To select a custom selection of JSF C++ rules, use an XML file.

Use the file `jsf_av_cpp.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *`polyspaceroot`*`\help\toolbox\bugfinder\examples\coding_standards_XML`. Here, *`polyspaceserverroot`* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2023a`. To update your script, see this table

| Option | Use Instead |
|---|---|
| `-jsf-coding-rules "custom_standard.conf"` | `-checkers-selection-file "custom_standard.conf.xml" -jsf-coding-rules from-file` |

See:

- “Configure Coding Rules Checking”
- “Setting Checkers in Polyspace as You Code”

### Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For a full list of rule IDs and section names, see:

- “AUTOSAR C++14 Rules”
- “CERT C Rules and Recommendations”
- “CERT C++ Rules”
- “ISO/IEC TS 17961 Rules”
- “Custom Coding Rules”
- “JSF C++ Rules”
- “MISRA C:2004 Rules”
- “MISRA C:2012 Directives and Rules”
- “MISRA C++:2008 Rules”
- “Guidelines”

**R2019b: Use of text format for coding rules file will not be supported**
*Warns starting in R2019b*

Since R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file. If you use a text file, Polyspace issues a warning and converts the text file to XML.

## See Also

`Do not generate results for (-do-not-generate-results-for)`

### Topics

"Specify Polyspace Analysis Options"
"Check for and Review Coding Standard Violations"
"JSF C++ Rules"

# Check AUTOSAR C++ 14 (`-autosar-cpp14`)

Check for violations of AUTOSAR C++ 14 rules

## Description

*This option affects Bug Finder only.*

Specify whether to check for violations of AUTOSAR C++ 14. Each value of the option corresponds to a subset of guidelines to check.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-178 for other options that you must also enable.

**Command line and options file**: Use the option `-autosar-cpp14`. See “Command-Line Information” on page 2-178.

### Why Use This Option

Use this option to specify the subset of AUTOSAR C++ 14 rules to check for[5].

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding standard violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

## Settings

**Default:** `all`

`all`

Check for violations of all AUTOSAR C++ 14 rules supported by Polyspace.

See “AUTOSAR C++14 Rules”.

`required`

Check for violations of *required* rules.

These rules are mandatory requirements placed on your code. This categorization of rules comes from the AUTOSAR C++14 guidelines.

`automated`

Check for violations of *automated* rules.

You can automatically enforce these rules by means of static analysis. This categorization of rules comes from the AUTOSAR C++14 guidelines.

[5] The Polyspace checkers for AUTOSAR C++14 rules supports AUTOSAR C++14 release 18-10 (October 2018). Out of 397 rules from the standard, 308 rules are supported.

Note that all rules in the required category might not be present in the automated category. For rules that AUTOSAR C++14 considers as non-automated, Bug Finder shows only a subset of actual rule violations.

from-file

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click Edit, then select the rules you want to check for this coding standard from the right pane of the **Checkers selection** window. Save the file.

To use or update an existing configuration file, in the **Checkers selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to from-file, enable Set checkers by file (-checkers-selection-file).

## Dependencies

- This option is available only if you set Source code language (-lang) to CPP or C-CPP.

## Command-Line Information

Use the command -autosar-cpp14 in the command line to check for violations of AUTOSAR C++ 14 rules.
**Parameter:** -autosar-cpp14
**Value:** all | required | automated | from-file
**Example (Bug Finder):** polyspace-bug-finder -lang cpp -sources *file_name* -autosar-cpp14 required
**Example (Bug Finder Server):** polyspace-bug-finder-server -lang cpp -sources *file_name* -autosar-cpp14 required

Alternatively, enable all or specific AUTOSAR C++ 14 rules through a checkers activation XML file. See -checkers-activation-file.

## See Also

Do not generate results for (-do-not-generate-results-for)

### Topics

"Specify Polyspace Analysis Options"
"Check for and Review Coding Standard Violations"
"AUTOSAR C++14 Rules"

# Check SEI CERT-C (-cert-c)

Check for violations of CERT C rules and recommendations

## Description

*This option affects Bug Finder only.*

Specify whether to check for violations of CERT C rules and recommendations. Each value of the option corresponds to a subset of the coding standard to check.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See "Dependencies" on page 2-186 for other options that you must also enable.

**Command line and options file**: Use the option `-cert-c`. See "Command-Line Information" on page 2-186.

### Why Use This Option

Use this option to specify the subset of CERT C rules and recommendations to check in your code.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding standard violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

## Settings

**Default:** `all`

`all-rules`

Check for violations of CERT C rules only.

See the CERT C website for an explanation of the difference between rules and recommendations.

**List of CERT-C rules that Polyspace checks when you use `all-rules`**

| |
|---|
| CERT C: Rule ARR30-C |
| CERT C: Rule ARR32-C |
| CERT C: Rule ARR36-C |
| CERT C: Rule ARR37-C |
| CERT C: Rule ARR38-C |
| CERT C: Rule ARR39-C |
| CERT C: Rule CON30-C |
| CERT C: Rule CON31-C |

| |
|---|
| CERT C: Rule CON32-C |
| CERT C: Rule CON33-C |
| CERT C: Rule CON35-C |
| CERT C: Rule CON36-C |
| CERT C: Rule CON37-C |
| CERT C: Rule CON40-C |
| CERT C: Rule CON41-C |
| CERT C: Rule CON43-C |
| CERT C: Rule DCL30-C |
| CERT C: Rule DCL31-C |
| CERT C: Rule DCL36-C |
| CERT C: Rule DCL37-C |
| CERT C: Rule DCL38-C |
| CERT C: Rule DCL39-C |
| CERT C: Rule DCL40-C |
| CERT C: Rule DCL41-C |
| CERT C: Rule ENV30-C |
| CERT C: Rule ENV31-C |
| CERT C: Rule ENV32-C |
| CERT C: Rule ENV33-C |
| CERT C: Rule ENV34-C |
| CERT C: Rule ERR30-C |
| CERT C: Rule ERR32-C |
| CERT C: Rule ERR33-C |
| CERT C: Rule ERR34-C |
| CERT C: Rule EXP30-C |
| CERT C: Rule EXP32-C |
| CERT C: Rule EXP33-C |
| CERT C: Rule EXP34-C |
| CERT C: Rule EXP35-C |
| CERT C: Rule EXP36-C |
| CERT C: Rule EXP37-C |
| CERT C: Rule EXP39-C |
| CERT C: Rule EXP40-C |
| CERT C: Rule EXP42-C |
| CERT C: Rule EXP43-C |
| CERT C: Rule EXP44-C |

| |
|---|
| CERT C: Rule EXP45-C |
| CERT C: Rule EXP46-C |
| CERT C: Rule EXP47-C |
| CERT C: Rule FIO30-C |
| CERT C: Rule FIO32-C |
| CERT C: Rule FIO34-C |
| CERT C: Rule FIO37-C |
| CERT C: Rule FIO38-C |
| CERT C: Rule FIO39-C |
| CERT C: Rule FIO40-C |
| CERT C: Rule FIO41-C |
| CERT C: Rule FIO42-C |
| CERT C: Rule FIO44-C |
| CERT C: Rule FIO45-C |
| CERT C: Rule FIO46-C |
| CERT C: Rule FIO47-C |
| CERT C: Rule FLP30-C |
| CERT C: Rule FLP32-C |
| CERT C: Rule FLP34-C |
| CERT C: Rule FLP36-C |
| CERT C: Rule FLP37-C |
| CERT C: Rule INT30-C |
| CERT C: Rule INT31-C |
| CERT C: Rule INT32-C |
| CERT C: Rule INT33-C |
| CERT C: Rule INT34-C |
| CERT C: Rule INT35-C |
| CERT C: Rule INT36-C |
| CERT C: Rule MEM30-C |
| CERT C: Rule MEM31-C |
| CERT C: Rule MEM33-C |
| CERT C: Rule MEM34-C |
| CERT C: Rule MEM35-C |
| CERT C: Rule MEM36-C |
| CERT C: Rule MSC30-C |
| CERT C: Rule MSC32-C |
| CERT C: Rule MSC33-C |

| |
|---|
| CERT C: Rule MSC37-C |
| CERT C: Rule MSC38-C |
| CERT C: Rule MSC39-C |
| CERT C: Rule MSC40-C |
| CERT C: Rule POS30-C |
| CERT C: Rule POS33-C (deprecated) |
| CERT C: Rule POS34-C |
| CERT C: Rule POS35-C |
| CERT C: Rule POS36-C |
| CERT C: Rule POS37-C |
| CERT C: Rule POS38-C |
| CERT C: Rule POS39-C |
| CERT C: Rule POS44-C |
| CERT C: Rule POS48-C |
| CERT C: Rule POS49-C |
| CERT C: Rule POS51-C |
| CERT C: Rule POS52-C |
| CERT C: Rule POS54-C |
| CERT C: Rule PRE30-C |
| CERT C: Rule PRE31-C |
| CERT C: Rule PRE32-C |
| CERT C: Rule SIG30-C |
| CERT C: Rule SIG31-C |
| CERT C: Rule SIG34-C |
| CERT C: Rule SIG35-C |
| CERT C: Rule STR30-C |
| CERT C: Rule STR31-C |
| CERT C: Rule STR32-C |
| CERT C: Rule STR34-C |
| CERT C: Rule STR37-C |
| CERT C: Rule STR38-C |
| CERT C: Rule WIN30-C |

`publish-2016`

Check for violations of CERT C rules only, as defined in the 2016 edition of the **SEI CERT C Coding Standard**.

See the CERT C website for an explanation of the difference between rules and recommendations.

### List of CERT-C rules that Polyspace checks when you use `publish-2016`

| |
|---|
| CERT C: Rule ARR30-C |
| CERT C: Rule ARR32-C |
| CERT C: Rule ARR36-C |
| CERT C: Rule ARR37-C |
| CERT C: Rule ARR38-C |
| CERT C: Rule ARR39-C |
| CERT C: Rule CON30-C |
| CERT C: Rule CON31-C |
| CERT C: Rule CON32-C |
| CERT C: Rule CON33-C |
| CERT C: Rule CON35-C |
| CERT C: Rule CON36-C |
| CERT C: Rule CON37-C |
| CERT C: Rule CON40-C |
| CERT C: Rule CON41-C |
| CERT C: Rule DCL30-C |
| CERT C: Rule DCL31-C |
| CERT C: Rule DCL36-C |
| CERT C: Rule DCL37-C |
| CERT C: Rule DCL38-C |
| CERT C: Rule DCL39-C |
| CERT C: Rule DCL40-C |
| CERT C: Rule DCL41-C |
| CERT C: Rule ENV30-C |
| CERT C: Rule ENV31-C |
| CERT C: Rule ENV32-C |
| CERT C: Rule ENV33-C |
| CERT C: Rule ENV34-C |
| CERT C: Rule ERR30-C |
| CERT C: Rule ERR32-C |
| CERT C: Rule ERR33-C |
| CERT C: Rule EXP30-C |
| CERT C: Rule EXP32-C |
| CERT C: Rule EXP33-C |
| CERT C: Rule EXP34-C |
| CERT C: Rule EXP35-C |

| |
|---|
| CERT C: Rule EXP36-C |
| CERT C: Rule EXP37-C |
| CERT C: Rule EXP39-C |
| CERT C: Rule EXP40-C |
| CERT C: Rule EXP42-C |
| CERT C: Rule EXP43-C |
| CERT C: Rule EXP44-C |
| CERT C: Rule EXP45-C |
| CERT C: Rule EXP46-C |
| CERT C: Rule FIO30-C |
| CERT C: Rule FIO32-C |
| CERT C: Rule FIO34-C |
| CERT C: Rule FIO37-C |
| CERT C: Rule FIO38-C |
| CERT C: Rule FIO39-C |
| CERT C: Rule FIO40-C |
| CERT C: Rule FIO41-C |
| CERT C: Rule FIO42-C |
| CERT C: Rule FIO44-C |
| CERT C: Rule FIO45-C |
| CERT C: Rule FIO46-C |
| CERT C: Rule FIO47-C |
| CERT C: Rule FLP30-C |
| CERT C: Rule FLP32-C |
| CERT C: Rule FLP34-C |
| CERT C: Rule FLP36-C |
| CERT C: Rule FLP37-C |
| CERT C: Rule INT30-C |
| CERT C: Rule INT31-C |
| CERT C: Rule INT32-C |
| CERT C: Rule INT33-C |
| CERT C: Rule INT34-C |
| CERT C: Rule INT35-C |
| CERT C: Rule INT36-C |
| CERT C: Rule MEM30-C |
| CERT C: Rule MEM31-C |
| CERT C: Rule MEM33-C |

| |
|---|
| `CERT C: Rule MEM34-C` |
| `CERT C: Rule MEM35-C` |
| `CERT C: Rule MEM36-C` |
| `CERT C: Rule MSC30-C` |
| `CERT C: Rule MSC32-C` |
| `CERT C: Rule MSC33-C` |
| `CERT C: Rule MSC37-C` |
| `CERT C: Rule MSC38-C` |
| `CERT C: Rule MSC39-C` |
| `CERT C: Rule MSC40-C` |
| `CERT C: Rule PRE30-C` |
| `CERT C: Rule PRE31-C` |
| `CERT C: Rule PRE32-C` |
| `CERT C: Rule SIG30-C` |
| `CERT C: Rule SIG31-C` |
| `CERT C: Rule SIG34-C` |
| `CERT C: Rule SIG35-C` |
| `CERT C: Rule STR30-C` |
| `CERT C: Rule STR31-C` |
| `CERT C: Rule STR32-C` |
| `CERT C: Rule STR34-C` |
| `CERT C: Rule STR37-C` |
| `CERT C: Rule STR38-C` |

`all`

Check for violations of all CERT C rules and recommendations supported by Polyspace.

See “CERT C Rules and Recommendations”.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click Edit, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Checkers selection** window. Save the file.

To use or update an existing configuration file, in the **Checkers selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

## Dependencies

- This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

  For projects with mixed C and C++ code, the SEI CERT-C checker analyzes only `.c` files.

## Command-Line Information

Use the command `-cert-c` in the command line to check for violations of CERT C rules and recommendations.
**Parameter:** `-cert-c`
**Value:** `all-rules` | `publish-2016` | `all` | `from-file`
**Example (Bug Finder):** `polyspace-bug-finder -lang c -sources` *`file_name`* `-cert-c all-rules`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -lang c -sources` *`file_name`* `-cert-c all-rules`

Alternatively, enable all or specific CERT C rules and recommendations through a checkers activation XML file. See `-checkers-activation-file`.

## See Also

`Do not generate results for (-do-not-generate-results-for)`

**Topics**
"Specify Polyspace Analysis Options"
"Check for and Review Coding Standard Violations"
"CERT C Rules and Recommendations"

# Check CWE (-cwe)

Check for weaknesses in your code that are enumerated in CWE

## Description

*This option affects Bug Finder only.*

Specify whether to check for common weaknesses enumerated in CWE. Each value of the option corresponds to a subset of the coding standard to check.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-186 for other options that you must also enable.

**Command line and options file**: Use the option `-cwe`. See “Command-Line Information” on page 2-186.

### Why Use This Option

Use this option to specify the subset of common weaknesses to check in your code.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding standard violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

## Settings

**Default:** `all`

`cwe-658-659`

Check for violations of a subset of rules for weaknesses that are specific to C (CWE-658) and C++ (CWE-659) software.

See section *CWE Rules Specific to C and C++ (CWE 658 and CWE 659)* of the “Common Weakness Enumeration (CWE)” category.

`all`

Check for violations of all CWE rules supported by Polyspace.

See section *CWE Rules* of the “Common Weakness Enumeration (CWE)” category.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click Edit, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Checkers selection** window. Save the file.

To use or update an existing configuration file, in the **Checkers selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

## Command-Line Information

Use the option `-cwe` in the command line to check for common weaknesses enumerated by CWE.
**Parameter:** `-cwe`
**Value:** `all` | `cwe-658-659` | `from-file`
**Example (Bug Finder):** `polyspace-bug-finder -lang c -sources` *`file_name`* `-cwe all`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -lang c -sources` *`file_name`* `-cwe all`

Alternatively, enable all or specific CWE rules by using a checkers activation XML file. See `-checkers-activation-file`.

## Tips

Some CWE rule violations that you checked for with a Polyspace product version R2022b or earlier might not be available in version R2023a or later when you use the option `-cwe`. You can use a mapping between CWE rules and Polyspace defect checkers to extend the Polyspace CWE coverage and check for these additional CWE rule violations. See “Extend CWE Coding Standard Coverage Using Polyspace Defect Checkers”.

# Version History

**Introduced in R2023a**

## See Also

`Do not generate results for (-do-not-generate-results-for)`

### Topics

“Specify Polyspace Analysis Options”
“Check for and Review Coding Standard Violations”
“Common Weakness Enumeration (CWE)”

# Check SEI CERT-C++ (`-cert-cpp`)

Check for violations of CERT C++ rules

## Description

*This option affects Bug Finder only.*

Specify whether to check for violations of CERT C++ rules.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-190 for other options that you must also enable.

**Command line and options file**: Use the option `-cert-cpp`. See “Command-Line Information” on page 2-190.

### Why Use This Option

Use this option to specify the subset of CERT C++ rules to check in your code.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding standard violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

## Settings

**Default:** `all`

`all`

Check for violations of all CERT C++ rules supported by Polyspace.

See “CERT C++ Rules”.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click Edit, then select the rules you want to check for this coding standard from the right pane of the **Checkers selection** window. Save the file.

To use or update an existing configuration file, in the **Checkers selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

## Dependencies

- This option is available only if you set to CPP or C-CPP.

  For projects with mixed C and C++ code, the SEI CERT-C++ checker analyzes only .cpp files.

## Command-Line Information

Use the command -cert-cpp in the command line to check for violations of CERT C++ rules.
**Parameter:** -cert-cpp
**Value:** all | from-file |
**Example (Bug Finder):** polyspace-bug-finder -lang cpp -sources *file_name* -cert-cpp all
**Example (Bug Finder Server):** polyspace-bug-finder-server -lang cpp -sources *file_name* -cert-cpp all

Alternatively, enable all or specific CERT C++ rules through a checkers activation XML file. See -checkers-activation-file.

## See Also

Do not generate results for (-do-not-generate-results-for)

### Topics

"Specify Polyspace Analysis Options"
"Check for and Review Coding Standard Violations"
"CERT C++ Rules"

# Check ISO/IEC TS 17961 (-iso-17961)

Check for violations of ISO/IEC TS 17961 rules

## Description

*This option affects Bug Finder only.*

Specify whether to check for violations of ISO/IEC TS 17961 rules.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See “Dependencies” on page 2-192 for other options that you must also enable.

**Command line and options file**: Use the option `-iso-17961`. See “Command-Line Information” on page 2-192.

### Why Use This Option

Use this option to specify the subset of ISO/IEC TS 17961 rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding standard violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

## Settings

**Default:** `all`

`decidable`

Check for violations of *decidable* rules. Violations of these rules depend only on compile-time static properties, for instance object type or scope of identifiers.

`all`

Check for violations of all ISO/IEC TS 17961 rules Polyspace supports.

See “ISO/IEC TS 17961 Rules”.

`from-file`

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click Edit, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Checkers selection** window. Save the file.

To use or update an existing configuration file, in the **Checkers selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

## Dependencies

- This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

## Command-Line Information

Use the command `-iso-17961` in the command line to check for violations of ISO/IEC TS 17961 rules.
**Parameter:** `-iso-17961`
**Value:**`decidable` | `all` | `from-file`
**Example (Bug Finder):** `polyspace-bug-finder -lang c -sources` *`file_name`* `-iso-17961 decidable`
**Example:** `polyspace-bug-finder-server -lang c -sources` *`file_name`* `-iso-17961 decidable`
**Example:** `polyspace-code-prover-server -lang c -sources` *`file_name`* `-iso-17961 decidable`

Alternatively, enable all or specific ISO/IEC TS 17961 rules through a checkers activation XML file. See `-checkers-activation-file`.

## See Also

`Do not generate results for (-do-not-generate-results-for)`

### Topics

"Specify Polyspace Analysis Options"
"Check for and Review Coding Standard Violations"
"ISO/IEC TS 17961 Rules"

# Check guidelines (`-guidelines`)

Check for violations of Guidelines

## Description

*This option affects Bug Finder only.*

Specify whether to check for violations of Guidelines. Each option value corresponds to a subset of guidelines to check.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

**Command line and options file**: Use the option `-guidelines`. See “Command-Line Information” on page 2-194.

### Why Use This Option

Guidelines are customizable checkers that check for violation of coding best practices. You can use this option to specify the subset of Guidelines rules that matches your requirements.

After analysis, the **Results List** pane lists the violations. On the **Source** pane, for every violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

## Settings

`all`

Check for violations of all Guidelines rules.

See “Guidelines”.

`his`

Check for violations of software complexity metrics standards specified in the Hersteller Initiative Software (HIS) standard. See “HIS Code Complexity Metrics”.

The HIS standard recommends specific thresholds for a subset of the software complexity checkers. When you use the input `his`, Polyspace activates this subset of software complexity checkers and uses their HIS recommended threshold. Polyspace raises a violation if a software complexity metric exceeds the HIS recommended threshold.

`from-file`

Specify an XML file where you configure a custom selection of Guidelines checkers that have specific thresholds. When running an analysis on Polyspace Desktop, create or edit an XML file that contains your checker configuration by using the Desktop User Interface. When running an analysis on Polyspace Server, edit an existing checker selection file. When running an analysis on Polyspace as You Code, create or edit an XML file that contains your checker configuration by using the Checkers Selection User interface.

When using the value `from-file`, use the option `Set checkers by file (-checkers-selection-file)` to specify the checker selection XML file.

## Tips

- When using an options file, you can activate the guideline checkers and specify a checkers selection file to modify their thresholds. For instance, in an options file, input:

  ```
  -guidelines from-file
  -checkers-selection-file selection_xml
  ```

  You can also use the preceding options together in the command line.
- Alternatively, you can specify a checkers activation file by using the command `-checkers-activation-file`. You do not need to specify the specific options for defects and coding standards. For instance, in the command line or in an options file, input:

  ```
  -checkers-activation-file activation_xml
  ```

  where the file `activation_xml` is a checkers activation file that is created by using the **Checkers Selection** User Interface. Specifying the option `-guidelines` is not needed if the Guidelines checkers are activated in the file `activation_xml`.

## Command-Line Information

Use the command `-guidelines` in the command line to check for violations of code complexity guidelines.
**Parameter:** `-guidelines`
**Value:** `all` | `his` | `from-file`
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-guidelines his`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-guidelines from-file -checkers-selection-file` *`path_to_xml`*

Alternatively, enable all or specific code complexity guidelines through a checkers activation XML file. See `-checkers-activation-file`.

# Version History

**Introduced in R2021a**

## See Also

`Set checkers by file (-checkers-selection-file)`

### Topics

"Specify Polyspace Analysis Options"
"Check for and Review Coding Standard Violations"
"Guidelines"
"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Calculate code metrics (-code-metrics)

Compute and display code complexity metrics

## Description

*This option applies to Bug Finder only. See Version History* on page 2-196.

Specify that Polyspace must compute and display code complexity metrics for your source code. The metrics include file metrics such as number of lines and function metrics such as cyclomatic complexity and estimated size of local variables.

For more information, see “Compute Code Complexity Metrics Using Polyspace”.

To maintain an acceptable level of software complexity during the development cycle, use the software complexity checkers. See “Reduce Software Complexity by Using Polyspace Checkers”.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

**Command line and options file**: Use the option `-code-metrics`. See “Command-Line Information” on page 2-196.

### Why Use This Option

By default, Polyspace does not calculate code complexity metrics. If you want these metrics in your analysis results, before running analysis, set this option.

High values of code complexity metrics can lead to obscure code and increase chances of coding errors. Additionally, if you run a Code Prover verification on your source code, you might benefit from checking your code complexity metrics first. If a function is too complex, attempts to verify the function can lead to a lot of unproven code. For information on how to cap your code complexity metrics, see “Compute Code Complexity Metrics Using Polyspace”.

## Settings

☑ On

Polyspace computes and displays code complexity metrics on the **Results List** pane.

□ Off (default)

Polyspace does not compute complexity metrics.

## Tips

If you want to compute only the code complexity metrics for your code, disable checking of defects. See `Find defects (-checkers)`.

## Command-Line Information

**Parameter:** `-code-metrics`
**Default:** Off
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-code-metrics`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-code-metrics`

# Version History

**R2022a: Higher and lower estimates of local variable size is not computed**
*Warns starting in R2022a*

If you compute code metrics by using Bug Finder, Polyspace issues a warning that these code metrics are not computed:

- `Higher Estimate of Size of Local Variables`
- `Lower Estimate of Size of Local Variables`

To compute these code metrics, use the option `Calculate stack usage (-stack-usage)`.

## See Also

**Topics**
"Compute Code Complexity Metrics Using Polyspace"
"Code Metrics"

# Find defects (`-checkers -disable-checkers`)

Enable or disable defect checkers

## Description

*This option affects a Bug Finder analysis only.*

Enable checkers for bugs/coding defects.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Bug Finder Analysis** node.

**Command line and options file**: Use the option `-checkers`. See “Command-Line Information” on page 2-198.

### Why Use This Option

The default set of checkers is designed to find the most meaningful bugs in most software development situations. If you have specific needs, enable or disable individual defect checkers. For instance, if you want to follow a specific security standard, choose a different subset of checkers.

## Settings

**Default:** `default`

`default`

A subset of defects defined by the software.

See “Polyspace Bug Finder Defects Checkers Enabled by Default”.

`all`

All defects.

For a list of all defects checkers, see “Defects”.

Note that the checkers `SQL injection` and `LDAP injection` are disabled even if you specify `all` for this option. You have to explicitly enable these checkers.

`custom`

Choose the defects you want to find by selecting categories of checkers or specific defects.

## Tips

You can use a spreadsheet to keep track of the defect checkers that you enable and add notes explaining why you do not enable the other checkers. A spreadsheet of checkers is provided in *`polyspaceroot`*`\polyspace\resources`. Here, *`polyspaceroot`* is the Polyspace installation folder, such as `C:\Program Files\Polyspace\R2023a`.

## Command-Line Information

Use the command `-checkers` in the command line to enable or disable defect checkers.

Regardless of order, the shell script processes the `-checkers` option, and then `-disable-checkers` option.

For the command-line parameters values, see "Short Names of Bug Finder Defect Groups and Defect Checkers".
**Parameter:** `-checkers`
**Value:** `default | all | none|"Short Names of Bug Finder Defect Groups and Defect Checkers"`
**Default:** `default`
**Parameter:** `-disable-checkers`
**Value:** `"Short Names of Bug Finder Defect Groups and Defect Checkers"`
**Example 1 (Bug Finder):** `polyspace-bug-finder -sources` *`filename`* `-checkers numerical,data_flow -disable-checkers FLOAT_ZERO_DIV`
**Example 2 (Bug Finder):** `polyspace-bug-finder -sources` *`filename`* `-checkers default -disable-checkers concurrency,dead_code`
**Example 1 (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`filename`* `-checkers numerical,data_flow -disable-checkers FLOAT_ZERO_DIV`
**Example 2 (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`filename`* `-checkers default -disable-checkers concurrency,dead_code`

Alternatively, activate the defect checkers in an activation XML file. See `-checkers-activation-file`.

# Version History

### R2023a: The value `CWE` will be removed in a future release

*Warns starting in R2023a*

The value `CWE` for the option `Find defects (-checkers -disable-checkers)` will be removed in a future release. Polyspace emits a warning if you use the option `-checkers CWE` at the command line and continues the analysis using option `-cwe all`. To check your code for CWE violations, use the option `Check CWE (-cwe)` instead.

### R2023a: The value `data_race_all` is removed

*Errors starting in R2023a*

The value `data_race_all` for the option `Find defects (-checkers -disable-checkers)` is removed. Using this option results in an error. To detect data races in atomic operations, use the analysis option `-detect-atomic-data-race`. See:

- `-detect-atomic-data-race`
- "Extend Data Race Checkers to Atomic Operations"

If you use `-checkers all` to run a Bug Finder analysis, you might see a reduced number of defects because this command no longer checks for data races in atomic operations. To mimic the result of `-checkers all` in previous releases, specify the option `-detect-atomic-data-race`.

To use `-detect-atomic-data-race` from the Polyspace user interface, in the **Advanced Settings** node, specify the option in the **Other** field.

## See Also

"Defects" | `Set checkers by file (-checkers-selection-file)` | `-checkers-activation-file`

### Topics

"Specify Polyspace Analysis Options"
"Short Names of Bug Finder Defect Groups and Defect Checkers"
"Bug Finder Defect Groups"

# Run stricter checks considering all values of system inputs (-checks-using-system-input-values)

Enable stricter checks and provide examples of values that lead to detected defect

## Description

*This option affects a Bug Finder analysis only. This option is not available for code generated from MATLAB code or Simulink models.*

Enable a stricter analysis mode for a subset of numerical, static memory and data flow related defect and coding standard checkers. If you enable this option:

- The analysis considers all possible values of system inputs when checking for issues.
- When an issue is detected, the analysis provides one example input value that leads to the issue.

For each function `foo` that you specify with `Consider inputs to these functions (-system-inputs-from)`, the following are considered as system inputs:

- Each argument of `foo`.
- Each read of a global variable by `foo` or one of its callees.

  For the `main()` function, the analysis assumes that the global variables are initialized with value 0.
- Each read of a volatile variable by `foo` or one of its callees.
- Each return of a stubbed function. a Bug Finder analysis stubs a function if you do not provide the body of the function in your source code.
- Each read from an absolute address, such as:

  ```
  int value = *((int*)0x1234);
  ```

For information on checkers affected by this option, see “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

You can view examples of values that lead to the detected defects in the **Events** column of the **Results Details** pane on the desktop interface or the Polyspace Access web interface.

### Set Option

**User interface** (desktop products only): In the **Configuration** pane, the option is on the **Bug Finder Analysis** node. See “Dependencies” on page 2-201 for other options that you must also enable.

**Command line and options file**: Use the option `-checks-using-system-input-values`. See “Command-Line Information” on page 2-202.

### Why Use This Option

The default Bug Finder analysis does not flag defects that are caused by specific values of unknown inputs. Since the inputs might be bounded or initialized in a source file that you are not analyzing, or the specific value causing a defect might not occur in practice, the default analysis behavior helps to minimize false positives.

Enable this option to run a stricter analysis on a function whose system inputs might cause sporadic run-time errors during execution. Using this option might result in a longer analysis time.

## Settings

☑ On

Polyspace considers all possible values of system inputs for a subset of numerical and static memory defect checkers and provides examples of values that lead to detected defects.

○ Off (default)

Polyspace considers possible values of a system input only if the input is bounded by constraints in your code such as `assert` or `if`. The analysis provides no examples of values that lead to detected defects.

## Dependencies

- In the desktop interface, this option is enabled only if you enable `Find defects (-checkers)`.
- This option is ignored if you enable `Use fast analysis mode for Bug Finder (-fast-analysis)`.

## Tips

- If you set external constraints on global variables, the analysis shows examples of global variable values causing defects only within these constraints. See `Constraint setup (-data-range-specifications)`.
- If the input is a pointer `p`, the analysis assumes that the pointer is not null and can be safely dereferenced. The example value of the input causing a defect is the value of `*p`. This value is represented as an array in the **Results Details** pane. For instance, in this code snippet:

  ```
  void func(int* x){
      int tmp= *(x+3);

      if(1/(tmp-4))
          return;
  }
  ```

  The example value of the input causing a defect is {0,0,0,4}, where the array represents `*x`, `*(x+1)`, `*(x+2)`, and `*(x+3)`. The value `*(x+3)=4` causes a division by zero.
- The analysis treats these standard library functions that read values from external sources as stubbed functions.
  - `getchar`
  - `getc`
  - `fgetc`
  - `scanf`
- The stricter analysis considers all possible values of system inputs but it is not an exhaustive analysis. If Bug Finder cannot determine whether a particular input causes a defect, no defect is

shown. For more on exhaustive analysis, see “Differences Between Polyspace Bug Finder and Polyspace Code Prover”.

## Command-Line Information

**Parameter:** `-checks-using-system-input-values`
**Default**: Off
**Example (Bug Finder):** `polyspace-bug-finder -sources file_name -checkers numerical,static_memory -checks-using-system-input-values`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources file_name -checkers numerical,static_memory -checks-using-system-input-values`

# Version History

**Introduced in R2020a**

## See Also

`Consider inputs to these functions (-system-inputs-from)`

### Topics

“Specify Polyspace Analysis Options”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

# Consider inputs to these functions (-system-inputs-from)

Specify functions for which the analysis considers all possible input values

## Description

*This option affects a Bug Finder analysis only.*

Specify the functions in your code for which Polyspace considers all possible input values. For each function that you specify with this option, the analysis considers all possible values of these inputs:

- Each argument of the function.
- Each read of a global variable by the function or one of its callees.

  For the `main()` function, the analysis assumes that the global variables are initialized with value 0.
- Each read of a volatile variable by the function or one of its callees.
- Each return of a stubbed function. a Bug Finder analysis stubs a function if you do not provide the body of the function in your source code.
- Each read from an absolute address, such as:

  ```
  int value = *((int*)0x1234);
  ```

### Set Option

**User interface** (desktop products only): In the **Configuration** pane, the option is on the **Bug Finder Analysis** node. See “Dependencies” on page 2-204 for other options that you must also enable.

**Command line and options file**: Use the option `-system-inputs-from`. See “Command-Line Information” on page 2-204.

### Why Use This Option

By default, Polyspace considers all possible input values for the `main()` function and tasks, if any, or uncalled functions with at least one callee if your code has no `main()`. Depending on the issue that you are investigating by running the stricter checks, specify a different subset of functions to analyze.

## Settings

**Default:** `auto`

`auto`

Consider all possible values for inputs to `main()` function and tasks, if any. You specify tasks with these options.

- `Cyclic tasks (-cyclic-tasks)`
- `Tasks (-entry-points)`

- `Interrupts (-interrupts)`

When the analyzed code has no `main()`, the analysis considers all possible values for inputs to uncalled functions with at least one callee.

`uncalled`

Consider all possible values for inputs to all uncalled functions.

`all`

Consider all possible values for inputs to all functions.

`custom`

Enter function names or choose from a list.

- Click ✚ to add a field and enter the function name.
- Click 🔍 to list functions in your code. Choose functions from the list.

### Dependencies

This option is enabled only if you enable `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`.

### Tips

- The analysis treats these standard library functions that read values from external sources as stubbed functions.
  - `getchar`
  - `getc`
  - `fgetc`
  - `scanf`

### Command-Line Information

**Parameter:** `-system-inputs-from`
**Value:** `auto | uncalled | all | custom`
**Default**: `auto`
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-checks-using-system-input-values -system-inputs-from custom=`*`func1`*`,`*`func2`*
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-checks-using-system-input-values -system-inputs-from custom=`*`func1`*`,`*`func2`*

## Version History

**Introduced in R2020a**

### See Also

`Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`

**Topics**

"Specify Polyspace Analysis Options"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Class (`-class-analyzer`)

Specify classes that you want to verify

## Description

*This option affects a Code Prover analysis only.*

Specify classes that Polyspace uses to generate a `main`.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-206 for other options that you must also enable.

**Command line and options file**: Use the option `-class-analyzer`. See “Command-Line Information” on page 2-207.

### Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option and the option `Functions to call within the specified classes (-class-analyzer-calls)` to specify the class methods that the generated `main` must call. Unless a class method is called directly or indirectly from `main`, the software does not analyze the method.

## Settings

**Default**: `all`

`all`

To generate a `main` function, Polyspace uses all classes that have at least one method defined outside a header file. The generated `main` calls methods that you specify using the option `Functions to call within the specified classes (-class-analyzer-calls)`.

`none`

The generated `main` cannot call any class method.

`custom`

To generate a `main` function, Polyspace uses classes that you specify. The generated `main` calls methods from classes that you specify using the option `Functions to call within the specified classes (-class-analyzer-calls)`.

## Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a `main` function.

- `Source code language (-lang)` is set to `CPP` or `C-CPP`.
- `Verify module or library (-main-generator)` is selected.

## Tips

- If you select `none` for this option, Polyspace will not verify class methods that you do not call explicitly in your code.
- Polyspace does not verify templates that are not instantiated. To verify a class template, explicitly instantiate a class using the template. See "Template Classes" (Polyspace Code Prover).

## Command-Line Information

**Parameter:** `-class-analyzer`
**Value:** `all` | `none` | `custom=`*`class1`*`[,`*`class2`*`,...]`
**Default:** `all`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2`

## See Also

`Verify module or library (-main-generator)` | `Functions to call within the specified classes (-class-analyzer-calls)` | `Analyze class contents only (-class-only)` | `Skip member initialization check (-no-constructors-init-check)`

### Topics

"Specify Polyspace Analysis Options"
"Verify C++ Classes" (Polyspace Code Prover)

# Functions to call within the specified classes (-class-analyzer-calls)

Specify class methods that you want to verify

## Description

*This option affects a Code Prover analysis only.*

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify class methods that Polyspace uses to generate a `main`. The generated `main` can call static, public and protected methods in classes that you specify using the **Class** option.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-209 for other options that you must also enable.

**Command line and options file**: Use the option `-class-analyzer-calls`. See “Command-Line Information” on page 2-209.

### Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option and the option `Class (-class-analyzer)` to specify the class methods that the generated `main` must call. Unless a class method is called directly or indirectly from `main`, the software does not analyze the method.

## Settings

**Default**: `unused`

`all`

The generated `main` calls all public and protected methods. It does not call methods inherited from a parent class.

`all-public`

The generated `main` calls all public methods. It does not call methods inherited from a parent class.

`inherited-all`

The generated `main` calls all public and protected methods including those inherited from a parent class.

`inherited-all-public`

The generated `main` calls all public methods including those inherited from a parent class.

`unused`

The generated `main` calls public and protected methods that are not called in the code.

`unused-public`

The generated `main` calls public methods that are not called in the code. It does not call methods inherited from a parent class.

`inherited-unused`

The generated `main` calls public and protected methods that are not called in the code including those inherited from a parent class.

`inherited-unused-public`

The generated `main` calls public methods that are not called in the code including those inherited from a parent class.

`custom`

The generated `main` calls the methods that you specify.

Enter function names or choose from a list.

- Click ✚ to add a field and enter the function name.
- Click 🔍 to list functions in your code. Choose functions from the list.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

## Dependencies

You can use this option only if:

- `Source code language (-lang)` is set to `CPP` or `C-CPP`.
- `Verify module or library (-main-generator)` is selected.

## Command-Line Information

**Parameter:** `-class-analyzer-calls`
**Value:** `all | all-public | inherited-all | inherited-all-public | unused | unused-public | inherited-unused | inherited-unused-public | custom=`*`method1`*`[,`*`method2`*`,...]`
**Default:** `unused`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public`

## See Also

`Verify module or library (-main-generator)` | `Class (-class-analyzer)`

### Topics

"Specify Polyspace Analysis Options"

"Verify C++ Classes" (Polyspace Code Prover)

# Analyze class contents only (`-class-only`)

Do not analyze code other than class methods

## Description

*This option affects a Code Prover analysis only.*

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify that Polyspace must verify only methods of classes that you specify using the option `Class (-class-analyzer)`.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-211 for other options that you must also enable.

**Command line and options file**: Use the option `-class-only`. See “Command-Line Information” on page 2-212.

### Why Use This Option

Use this option to restrict the analysis to certain class methods only.

You specify these methods through the options:

- `Class (-class-analyzer)`
- `Functions to call within the specified classes (-class-analyzer-calls)`

When you analyze a module or library, Code Prover generates a `main` function if one does not exist. The `main` function calls class methods using these two options and functions that are not class methods using other options. Code Prover analyzes these methods and functions for robustness to all inputs. If you use this option, Code Prover analyzes the methods only.

## Settings

☑ On

Polyspace verifies the class methods only. It stubs functions out of class scope even if the functions are defined in your code.

□ Off (default)

Polyspace verifies functions out of class scope in addition to class methods.

## Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a `main` function.
- `Source code language (-lang)` is set to `CPP` or `C-CPP`.
- `Verify module or library (-main-generator)` is selected.

If you select this option, you must specify the classes using the `Class (-class-analyzer)` option.

## Tips

Use this option:

- For robustness verification of class methods. Unless you use this option, Polyspace verifies methods that you call in your code only for your input combinations.
- In case of scaling.

## Command-Line Information

**Parameter:** `-class-only`
**Default**: Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -class-only`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -class-only`

## See Also

`Verify module or library (-main-generator)` | `Class (-class-analyzer)` | `Functions to call within the specified classes (-class-analyzer-calls)`

### Topics

“Specify Polyspace Analysis Options”
“Verify C++ Classes” (Polyspace Code Prover)

# Initialization functions (-functions-called-before-main)

Specify functions that you want the generated main to call ahead of other functions

## Description

*This option affects a Code Prover analysis only.*

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify functions that you want the generated main to call ahead of other functions.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See "Dependencies" on page 2-214 for other options that you must also enable.

**Command line and options file**: Use the option -functions-called-before-main. See "Command-Line Information" on page 2-214.

### Why Use This Option

If you are verifying a module or library, Code Prover generates a main function if one does not exist. If a main exists, the analysis uses the existing main.

Use this option along with the option Functions to call (-main-generator-calls) to specify which functions the generated main must call. Unless a function is called directly or indirectly from main, the software does not analyze the function.

## Settings

**No Default**

Enter function names or choose from a list.

- Click ✚ to add a field and enter the function name.
- Click 🔍 to list functions in your code. Choose functions from the list.

If the function or method is not overloaded, specify the function name. Otherwise, specify the function prototype with arguments. For instance, in the following code, you must specify the prototypes func(int) and func(double).

```
int func(int x) {
 return(x * 2);
}
double func(double x) {
 return(x * 2);
}
```

For C++, if the function is:

- A class method: The generated `main` calls the class constructor before calling this function.
- Not a class method: The generated `main` calls this function before calling class methods.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::init(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::init()`.

## Dependencies

This option is enabled only if you select **Verify module or library** under **Code Prover Verification** and your code does not contain a `main` function.

## Tips

Although these functions are called ahead of other functions, they can be called in arbitrary order. If you want to call your initialization functions in a specific order, manually write a `main` function to call them.

## Command-Line Information

**Parameter:** `-functions-called-before-main`
**Value:** `*function1*[,*function2*[,...]]`
**No Default**
**Example 1 (Code Prover):** `polyspace-code-prover -sources *file_name* -main-generator -functions-called-before-main myfunc`
**Example 2 (Code Prover):** `polyspace-code-prover -sources *file_name* -main-generator -functions-called-before-main myClass::init(int)`
**Example 1 (Code Prover Server):** `polyspace-code-prover-server -sources *file_name* -main-generator -functions-called-before-main myfunc`
**Example 2 (Code Prover Server):** `polyspace-code-prover-server -sources *file_name* -main-generator -functions-called-before-main myClass::init(int)`

## See Also

`Verify module or library (-main-generator)` | `Variables to initialize (-main-generator-writes-variables)` | `Functions to call (-main-generator-calls)` | `Class (-class-analyzer)` | `Functions to call within the specified classes (-class-analyzer-calls)`

### Topics

"Specify Polyspace Analysis Options"
"Verify C Application Without main Function" (Polyspace Code Prover)
"Verify C++ Classes" (Polyspace Code Prover)

# Verify initialization section of code only (-init-only-mode)

Check initialization code alone for run-time errors and other issues

## Description

*This option affects a Code Prover analysis only.*

Specify that Polyspace must check only the section of code marked as initialization code for run-time errors and other issues.

To indicate the end of initialization code, you enter the line

```
#pragma polyspace_end_of_init
```

in the `main` function (only once). The initialization code starts from the beginning of `main` and continues up to this pragma.

Since compilers ignore unrecognized pragmas, the presence of this pragma does not affect program execution.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

**Command line and options file**: Use the option `-init-only-mode`. See “Command-Line Information” on page 2-217.

### Why Use This Option

Often, issues in the initialization code can invalidate the analysis of the remaining code. You can use this option to check the initialization code alone and fix the issues, and then disable this option to verify the remaining program.

For instance, in this example:

```
#include <limits.h>

int aVar;
const int aConst = INT_MAX;
int anotherVar;

int main() {
      aVar = aConst + 1;
#pragma polyspace_end_of_init
      anotherVar = aVar - 1;
      return 0;
}
```

the overflow in the line `aVar = aConst+1` must be fixed first before the value of `aVar` is used in subsequent code.

## Settings

☑ On

Polyspace checks the code from the beginning of `main` and continues up to the pragma `polyspace_end_of_init`.

○ Off (default)

Polyspace checks the complete application beginning from the `main` function.

## Dependencies

You can use this option and designate a section of code as initialization code only if:

- Your program contains a `main` function and you use the option `Verify whole application` (implicitly set by default at command line).
- You set `Source code language (-lang)` to `C`.

Note that the pragma must appear only once in the `main` function. The pragma can appear before or after variable declarations but must appear after type definitions (`typedef`-s).

You cannot use this option with the following options:

- `Verify files independently (-unit-by-unit)`
- `Show global variable sharing and usage only (-shared-variables-mode)`

## Tips

- Use this option along with the option `Check that global variables are initialized after warm reboot (-check-globals-init)` to thoroughly check the initialization code before checking the remaining program. If you use both options, the verification checks for the following:

  - Definite or possible run-time errors in the initialization code.
  - Whether all non-`const` global variables are initialized along all execution paths through the initialization code.
- Multitasking options are disabled if you check initialization code only because the initialization of global variables is expected to happen before the tasks (threads) begin. As a result, task bodies are not verified.

  See also “Multitasking” (Polyspace Code Prover).
- If you check initialization code only, the analysis truncates execution paths containing the pragma at the location of the pragma but continues to check other execution paths.

  For instance, in this example, the `pragma` appears in an `if` block. A red non-initialized variable check appears on the line `int a = var` because the path containing the initialization stops at the location of the pragma. On the only other remaining path that bypasses the `if` block, the variable `var` is not initialized.

  ```
  int var;

  int func();
  ```

```
int main() {
    int err = func();
    if(err) {
        var = 0;
 #pragma polyspace_end_of_init
    }
    int a = var;
    return 0;
}
```

To avoid these situations, try to place the pragma outside a block. See other suggestions for placement of the pragma in the reference for `Check that global variables are initialized after warm reboot (-check-globals-init)`.

- To determine the initialization of a structure, a regular Code Prover analysis only considers fields that are used.

  If you check initialization code only using this option, the analysis covers only a portion of the code and cannot determine if a variable is used beyond this portion. Therefore, the checks for initialization consider all structure fields, whether used or not.

### Command-Line Information

**Parameter:** `-init-only-mode`
**Default**: Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-init-only-mode`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-init-only-mode`

## Version History

**Introduced in R2020a**

### See Also

`Check that global variables are initialized after warm reboot (-check-globals-init)` | `Global variable not assigned a value in initialization code`

#### Topics

“Specify Polyspace Analysis Options”
“Code Prover Assumptions About Global Variable Initialization” (Polyspace Code Prover)

# Verify whole application

Stop verification if sources files are incomplete and do not contain a `main` function

## Description

*This option affects a Code Prover analysis only.*

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify that Polyspace verification must stop if a `main` function is not present in the source files.

If you select a Visual C++ setting for `Compiler (-compiler)`, you can specify which function must be considered as `main`. See `Main entry point (-main)`.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

**Command line and options file**: There is no corresponding command-line option. See “Command-Line Information” on page 2-218.

## Settings

● On

Polyspace verification stops if it does not find a `main` function in the source files.

○ Off (default)

Polyspace continues verification even when a `main` function is not present in the source files. If a `main` is not present, it generates a file `__polyspace_main.c` that contains a `main` function.

## Tips

If you use this option, your code must contain a `main` function. Otherwise you see the error:

```
Error: required main procedure not found
```

If your code does not contain a `main` function, use the option `Verify module or library (-main-generator)` to generate a `main` function.

## Command-Line Information

Unlike the user interface, by default, a verification from the command line stops if it does not find a `main` function in the source files. If you specify the option `-main-generator`, Polyspace generates a `main` if it cannot find one in the source files.

## See Also

`Verify module or library (-main-generator)` | `Show global variable sharing and usage only (-shared-variables-mode)`

### Topics

"Specify Polyspace Analysis Options"
"Verify C Application Without main Function" (Polyspace Code Prover)
"Verify C++ Classes" (Polyspace Code Prover)

# Show global variable sharing and usage only (`-shared-variables-mode`)

Compute global variable sharing and usage without running full analysis

## Description

*This option affects a Code Prover analysis only.*

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify this option to run a less extensive analysis that computes the global variable sharing and usage in your entire application. The analysis does not verify your code for run-time errors. The analysis results also include coding standards violations if you enable coding standards checking, and code metrics if you enable code metrics computation.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

**Command line and options file**: Use the option `-shared-variables-mode`. See "Command-Line Information" on page 2-221.

### Why Use This Option

You can see global variable sharing and usage without running a full analysis on your entire application that includes run-time error detection. Run-time error detection on an entire application can take a long time.

## Settings

☑ On

Polyspace computes global variable sharing and usage but does not verify your code for run-time errors.

○ Off (default)

Polyspace runs a full analysis on your code, including run-time error detection.

## Dependencies

- You can use this option only if your program contains a `main` function and you enable the option `Verify whole application` (implicitly set by default at command line).
- When you enable this option, you must also enable at least one of these options.
  - `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`
  - `Tasks (-entry-points)`

- `Cyclic tasks (-cyclic-tasks)`
- `Interrupts (-interrupts)`
- `ARXML files selection (-autosar-multitasking)`
- `OIL files selection (-osek-multitasking)`

## Tips

- After you analyze your complete application to see global variable sharing and usage, run a component-by-component Code Prover analysis to detect run-time errors.
- In the desktop product, you can see all read and write operations on global variables in the "Variable Access in Polyspace Desktop User Interface" (Polyspace Code Prover) pane.
- In this less extensive analysis mode, the analysis checks for most but not all coding standards violations, and computes most but not all code metrics.

## Command-Line Information

**Parameter:** `-shared-variables-mode`
**Default**: Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-shared-variables-mode -enable-concurrency-detection`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-shared-variables-mode -enable-concurrency-detection`

# Version History

**Introduced in R2019b**

## See Also

### Topics

"Specify Polyspace Analysis Options"

# Main entry point (-main)

Specify a Microsoft Visual C++ extensions of `main`

## Description

*This option affects a Code Prover analysis only.*

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify the function that you want to use as `main`. If the function does not exist, the verification stops with an error message. Use this option to specify Microsoft Visual C++ extensions of `main`.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-222 for other options that you must also enable.

**Command line and options file**: Use the option `-main`. See “Command-Line Information” on page 2-223.

## Settings

**Default:** `_tmain`

`_tmain`
: Use `_tmain` as entry point to your code.

`wmain`
: Use `wmain` as entry point to your code.

`_tWinMain`
: Use `_tWinMain` as entry point to your code.

`wWinMain`
: Use `wWinMain` as entry point to your code.

`WinMain`
: Use `WinMain` as entry point to your code.

`DllMain`
: Use `DllMain` as entry point to your code.

## Dependencies

This option is enabled only if you:

- Set `Source code language (-lang)` to `CPP`.
- Select `Verify whole application`.

## Command-Line Information

**Parameter:** `-main`
**Value:** `_tmain` | `wmain` | `_tWinMain` | `wWinMain` | `WinMain` | `DllMain`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-compiler visual14.0 -main _tmain`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-compiler visual14.0 -main _tmain`

## See Also

`Verify whole application` | `Verify module or library (-main-generator)`

### Topics

"Specify Polyspace Analysis Options"
"Verify C Application Without main Function" (Polyspace Code Prover)
"Verify C++ Classes" (Polyspace Code Prover)

# Functions to call (`-main-generator-calls`)

Specify functions that you want the generated `main` to call after the initialization functions

## Description

*This option affects a Code Prover analysis only.*

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify functions that you want the generated `main` to call. The `main` calls these functions after the ones you specify through the option `Initialization functions (-functions-called-before-main)`.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-225 for other options that you must also enable.

**Command line and options file**: Use the option `-main-generator-calls`. See “Command-Line Information” on page 2-225.

### Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option along with the option `Initialization functions (-functions-called-before-main)` to specify which functions the generated `main` must call. Unless a function is called directly or indirectly from `main`, the software does not analyze the function.

## Settings

**Default:** `unused`

`none`

The generated `main` does not call any function.

`unused`

The generated `main` calls only those functions that are not called in the source code. It does not call inlined functions.

`all`

The generated `main` calls all functions except inlined ones.

`custom`

The generated `main` calls functions that you specify.

Enter function names or choose from a list.

- Click ✚ to add a field and enter the function name.
- Click 🔍 to list functions in your code. Choose functions from the list.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

## Dependencies

This option is available only if you select `Verify module or library (-main-generator)`.

## Tips

- Select `unused` when you use **Code Prover Verification** > **Verify files independently**.
- If you want the generated `main` to call an inlined function, select `custom` and specify the name of the function.
- To verify a multitasking application without a main, select `none`.
- The generated `main` can call the functions in arbitrary order. If you want to call your functions in a specific order, manually write a `main` function to call them.
- To specify instantiations of templates as arguments, run analysis once with the option argument `all`. Search for the template name in the analysis log and use the template name as it appears in the analysis log for the option argument.

  For instance, to specify this template function instantiation as option argument:

  ```
  template <class T>
  T GetMax (T a, T b) {
    T result;
    result = (a>b)? a : b;
    return (result);
  }
  template int GetMax<int>(int, int); // explicit instantiation
  ```

  Run an analysis with the option `-main-generator-calls all`. Search for `getMax` in the analysis log. You see the function format:

  ```
  T1 getMax<int>(T1, T1)
  ```

  To call only this template instantiation, remove the space between the arguments and use the option:

  ```
  -main-generator-calls custom="T1 getMax<int>(T1,T1)"
  ```

## Command-Line Information

**Parameter:** `-main-generator-calls`
**Value:** `none` | `unused` | `all` | `custom=`*`function1`*`[,`*`function2`*`[,...]]`
**Default:** `unused`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-main-generator -main-generator-calls all`

**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -main-generator-calls all`

## See Also

`Verify module or library (-main-generator)` | `Initialization functions (-functions-called-before-main)` | `Class (-class-analyzer)` | `Functions to call within the specified classes (-class-analyzer-calls)`

### Topics

"Specify Polyspace Analysis Options"
"Verify C Application Without main Function" (Polyspace Code Prover)

# Variables to initialize (-main-generator-writes-variables)

Specify global variables that you want the generated `main` to initialize

## Description

*This option affects a Code Prover analysis only.*

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify global variables that you want the generated `main` to initialize. Polyspace considers these variables to have any value allowed by their type.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-228 for other options that you must also enable.

**Command line and options file**: Use the option `-main-generator-writes-variables`. See “Command-Line Information” on page 2-228.

### Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

A Code Prover analysis of a module without a `main` function makes some default assumptions about global variable initialization. The analysis assumes that global variables that are not explicitly initialized can have the full range of values allowed by their data types upon each entry into an uncalled function. For instance, in the example below, which does not have a `main` function, the variable `glob` is assumed to have *all* possible `int` values both in `foo` and `bar` (despite the modification in `foo`). The assumption is a conservative one since the call context of `foo` and `bar`, including which function gets called earlier, is not known.

```
int glob;

int foo() {
   int locFoo = glob;
   glob++;
   return locFoo;
}

int bar() {
   int locBar = glob;
   return locBar;
}
```

To implement this assumption, the generation `main` initializes such global variables to full-range values before calling each otherwise uncalled function. Use this option to modify this default assumption and implement a different initialization strategy for global variables.

## Settings

**Default:**

- C code — `public`
- C++ Code — `uninit`

`uninit`

C++ Only

The generated `main` only initializes global variables that you have not initialized during declaration.

`none`

The generated `main` does not initialize global variables.

Global variables are initialized according to the C/C+ standard. For instance, `int` or `char` variables are initialized to 0, `float` variables to 0.0, and so on.

`public`

The generated `main` initializes all global variables except those declared with keywords `static` and `const`.

`all`

The generated `main` initializes all global variables except those declared with keyword `const`.

`custom`

The generated `main` only initializes global variables that you specify. Click ✚ to add a field. Enter a global variable name.

## Dependencies

You can use this option only if the following are true:

- Your code does not contain a `main` function.
- `Verify module or library (-main-generator)` is selected.

The option is disabled if you enable the option `Ignore default initialization of global variables (-no-def-init-glob)`. Global variables are considered as uninitialized until you explicitly initialize them in the code.

## Tips

This option only affects global variables that are defined in the project. If a global variable is declared as `extern`, the analysis considers that the variable can have any value allowed by its data type, irrespective of the value of this option.

## Command-Line Information

**Parameter:** `-main-generator-writes-variables`
**Value:** `uninit | none` | `public` | `all` | `custom=`*`variable1`*`[,`*`variable2`*`[,...]]`
**Default:** (C) `public |` (C++) `uninit`

**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-main-generator -main-generator-writes-variables all`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -main-generator-writes-variables all`

## See Also

`Verify module or library (-main-generator)`

### Topics

“Specify Polyspace Analysis Options”
“Verify C Application Without main Function” (Polyspace Code Prover)

# Skip member initialization check (`-no-constructors-init-check`)

Do not check if class constructor initializes class members

## Description

*This option affects a Code Prover analysis only.*

Specify that Polyspace must not check whether each class constructor initializes all class members.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-230 for other options that you must also enable.

**Command line and options file**: Use the option `-no-constructors-init-check`. See “Command-Line Information” on page 2-231.

### Why Use This Option

Use this option to disable checks for initialization of class members in constructors.

## Settings

☑ On

Polyspace does not check whether each class constructor initializes all class members.

□ Off (default)

Polyspace checks whether each class constructor initializes all class members. It checks for initialization of:

- Integer types such as `int`, `char` and `enum`, both `signed` or `unsigned`.
- Floating-point types such as `float` and `double`.
- Pointers.

The analysis uses built-in functions `__ps_builtin_check_NIV()` and `__ps_builtin_check_NIP()` in the generated `main` to perform these checks. If the Code Prover analysis proves that a class member is not initialized in the constructor, you see a red check on this member in a call to the function `__ps_builtin_check_NIV()`. If the analysis cannot prove definite initialization or non-initialization of a class member in the constructor, you see an orange check.

## Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a `main` function.
- `Source code language (-lang)` is set to `CPP` or `C-CPP`.
- `Verify module or library (-main-generator)` is selected.

If you select this option, you must specify the classes using the`Class (-class-analyzer)` option.

## Command-Line Information

**Parameter:** `-no-constructors-init-check`
**Default**: Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -no-constructors-init-check`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -no-constructors-init-check`

## See Also

`Verify module or library (-main-generator)` | `Class (-class-analyzer)`

### Topics

"Specify Polyspace Analysis Options"
"Verify C++ Classes" (Polyspace Code Prover)

# Verify files independently (`-unit-by-unit`)

Verify each source file independently of other source files

## Description

*This option affects a Code Prover analysis only.*

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify that each source file must be verified independently of other source files. Each file is verified individually, independent of other files in the module. Verification results can be viewed for the entire project or for individual files.

After you open the verification result for one file, in the user interface of the Polyspace desktop products, you can see a summary of results for all files on the **Dashboard** pane. You can open the results for each file directly from this summary table.

Each result file (with name `ps_results.pscp`) is saved in a subfolder of the results folder. The subfolder has the same name as the source file being analyzed.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-232 for other options that you must also enable.

**Command line and options file**: Use the option `-unit-by-unit`. See “Command-Line Information” on page 2-233.

### Why Use This Option

There are many reasons you might want to verify each source file independently of other files.

For instance, if verification of a project takes very long, you can perform a file by file verification to identify which file is slowing the verification.

## Settings

☑ On

Polyspace creates a separate verification job for each source file.

□ Off (default)

Polyspace creates a single verification job for all source files in a module.

## Dependencies

This option is enabled only if you select `Verify module or library (-main-generator)`.

## Tips

- Code Prover requires a `main` function as the starting point of verification. In the file-by-file mode, because most files do not have a `main`, Code Prover generates a `main` function when required. By default, the generated `main` calls uncalled functions (uncalled non-private methods and out-of-class functions in C++). For more information, see:
  - “Verify C Application Without main Function” (Polyspace Code Prover)
  - “Verify C++ Classes” (Polyspace Code Prover)
- If you perform a file by file verification, you cannot specify multitasking options.
- If your verification for the entire project takes very long, perform a file by file verification. After the verification is complete for a file, you can view the results while other files are still being verified.
- You can generate a report of the verification results for each file or for all the files together. To generate a single report for all files, perform the report generation after verification (and not along with verification using analysis options).

  To generate a single report for all the files in the Polyspace user interface (desktop product only):

  1. Open the results for one file.
  2. Select **Reporting** > **Run Report**. Before generating the report, select the option **Generate a single report including all unit results**.

  If you use the product Polyspace Code Prover Server to run a verification, to generate a single report for all files:

  - Upload the results for all files to the Polyspace Access server.
  - Use the `polyspace-report-generator` command with option `-all-units` to generate a single report for all the files.
- When you perform a file-by-file verification, you can see many instances of unused variables. Some of these variables might be used in other files but show as unused in a file-by-file verification.

  If you want to ignore these results, use a review scope (named set of filters) that filters out unused variables. See “Filter and Group Results in Polyspace Desktop User Interface” (Polyspace Code Prover).

## Command-Line Information

**Parameter:** `-unit-by-unit`
**Default**: Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file1,file2,...`* `-unit-by-unit`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file1,file2,...`* `-unit-by-unit`

## See Also

`Common source files (-unit-by-unit-common-source)`

### Topics

“Specify Polyspace Analysis Options”

# Common source files (-unit-by-unit-common-source)

Specify files that you want to include with each source file during a file by file verification

## Description

*This option affects a Code Prover analysis only.*

*This option is not available for code generated from MATLAB code or Simulink models.*

For a file by file verification, specify files that you want to include with each source file verification. These files are compiled once, and then linked to each verification.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See “Dependencies” on page 2-234 for other options that you must also enable.

**Command line and options file**: Use the option `-unit-by-unit-common-source`. See “Command-Line Information” on page 2-235.

### Why Use This Option

There are many reasons you might want to verify each source file independently of other files. For instance, if verification of a project takes very long, you can perform a file by file verification to identify which file is slowing the verification.

If you perform a file by file verification, some of your files might be missing information present in the other files. Place the missing information in a common file and use this option to specify the file for verification. For instance, if multiple source files call the same function, use this option to specify a file that contains the function definition or a function stub. Otherwise, Polyspace uses its own stubs for functions that are called but not defined in the source files. The assumptions behind the Polyspace stubs can be broader than what you want, leading to orange checks.

## Settings

**No Default**

Click ✚ to add a field. Enter the full path to a file. Otherwise, use the 📁 button to navigate to the file location.

## Dependencies

This option is enabled only if you select `Verify files independently (-unit-by-unit)`.

## Command-Line Information

**Parameter:** `-unit-by-unit-common-source`
**Value:** `*file1*[,*file2*[,...]]`
**No Default**
**Example (Code Prover):** `polyspace-code-prover -sources *file_name* -unit-by-unit -unit-by-unit-common-source definitions.c`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources *file_name* -unit-by-unit -unit-by-unit-common-source definitions.c`

## See Also

`Verify files independently (-unit-by-unit)`

### Topics

"Specify Polyspace Analysis Options"

# Verify model generated code (`-main-generator`)

Specify that a `main` function must be generated if it is not present in source files

## Description

*This option is automatically set if you run Polyspace from Simulink or MATLAB on generated code. If you run Polyspace on generated code outside Simulink or MATLAB, set this option manually.*

Specify that Polyspace must generate a `main` function if it does not find one in the source files.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

**Command line and options file**: Use the option `-main-generator`. See “Command-Line Information” on page 2-236.

## Settings

This option is always enabled for code generated from models.

Polyspace generates a `main` function for the analysis. The generated `main` contains cyclic code that executes in a loop. The loop can run an unspecified number of times.

The `main` performs the following functions before the loop begins:

- Initializes variables specified by `Parameters (-variables-written-before-loop)`.
- Calls the functions specified by `Initialization functions (-functions-called-before-loop)`.

The `main` then performs the following functions in the loop:

- Calls the functions specified by `Step functions (-functions-called-in-loop)`.
- Writes to variables specified by `Inputs (-variables-written-in-loop)`.

Finally, the `main` calls the functions specified by `Termination functions (-functions-called-after-loop)`.

## Command-Line Information

**Parameter:** `-main-generator`
**Default:** Off
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-main-generator ...`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-main-generator ...`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-main-generator ...`

**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-main-generator ...`

## See Also

`Parameters (-variables-written-before-loop)` | `Initialization functions (-functions-called-before-loop)` | `Step functions (-functions-called-in-loop)` | `Inputs (-variables-written-in-loop)` | `Termination functions (-functions-called-after-loop)`

### Topics

"Configure Polyspace Options in Simulink"
"How Polyspace Analysis of Generated Code Works"

# Initialization functions (`-functions-called-before-loop`)

Specify functions that the generated `main` must call before the cyclic code loop

## Description

*This option is automatically set if you run Polyspace from Simulink or MATLAB on generated code. If you run Polyspace on generated code outside Simulink or MATLAB, set this option manually.*

Specify functions that the generated `main` must call before the cyclic code begins.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node. You see this option only if you open a project configuration from Simulink.

**Command line and options file**: Use the option `-functions-called-before-loop`. See “Command-Line Information” on page 2-238.

## Settings

**No Default** if you run the analysis outside Simulink. If you run from Simulink, the option uses the initialize functions from the generated code by default. See also “How Polyspace Analysis of Generated Code Works”.

Click ✚ to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::init(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::init()`.

## Tips

- If you specify a function for the option `Termination functions (-functions-called-after-loop)`, you cannot specify it for this option.

## Command-Line Information

**Parameter:** `-functions-called-before-loop`
**No Default**
**Value:** `*function1*[,*function2*[,...]]`
**Example (Bug Finder):** `polyspace-bug-finder -sources *file_name* -main-generator -functions-called-before-loop myfunc`
**Example (Code Prover):** `polyspace-code-prover -sources *file_name* -main-generator -functions-called-before-loop myfunc`

**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-main-generator -functions-called-before-loop myfunc`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -functions-called-before-loop myfunc`

## See Also

`Verify model generated code (-main-generator)` | `Step functions (-functions-called-in-loop)` | `Termination functions (-functions-called-after-loop)`

### Topics

"Configure Polyspace Options in Simulink"
"How Polyspace Analysis of Generated Code Works"

# Step functions (`-functions-called-in-loop`)

Specify functions that the generated `main` must call in the cyclic code loop

## Description

*This option is automatically set if you run Polyspace from Simulink or MATLAB on generated code. If you run Polyspace on generated code outside Simulink or MATLAB, set this option manually.*

Specify functions that the generated `main` must call in each cycle of the cyclic code.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node. You see this option only if you open a project configuration from Simulink.

**Command line and options file**: Use the option `-functions-called-in-loop`. See “Command-Line Information” on page 2-241.

## Settings

**Default:** `none` if you run the analysis outside Simulink. If you run from Simulink, the option uses the step functions from the generated code by default. See also “How Polyspace Analysis of Generated Code Works”.

`none`

The generated `main` does not call functions in the cyclic code.

`all`

The generated `main` calls all functions except inlined ones. If you specify certain functions for the options **Initialization functions** or **Termination functions**, the generated `main` does not call those functions in the cyclic code.

`custom`

The generated `main` calls functions that you specify. Click ✚ to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

## Tips

If you have specified a function for the option `Initialization functions (-functions-called-before-loop)` or `Termination functions (-functions-called-after-loop)`, to call it inside the cyclic code, use `custom` and specify the function name.

## Command-Line Information

**Parameter:** `-functions-called-in-loop`
**Value:** `none` | `all` | `custom=`*`function1`*`[,`*`function2`*`[,...]]`
**Default:** `none`
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-main-generator -functions-called-in-loop all`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-main-generator -functions-called-in-loop all`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-main-generator -functions-called-in-loop all`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -functions-called-in-loop all`

## See Also

`Verify model generated code (-main-generator)` | `Initialization functions (-functions-called-before-loop)` | `Termination functions (-functions-called-after-loop)`

### Topics

"Configure Polyspace Options in Simulink"
"How Polyspace Analysis of Generated Code Works"

# Termination functions (`-functions-called-after-loop`)

Specify functions that the generated `main` must call after the cyclic code loop

## Description

*This option is automatically set if you run Polyspace from Simulink or MATLAB on generated code. If you run Polyspace on generated code outside Simulink or MATLAB, set this option manually.*

Specify functions that the generated `main` must call after the cyclic code ends.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node. You see this option only if you open a project configuration from Simulink.

**Command line and options file**: Use the option `-functions-called-after-loop`. See “Command-Line Information” on page 2-242.

## Settings

**No Default** if you run the analysis outside Simulink. If you run from Simulink, the option uses the terminate functions from the generated code by default. See also “How Polyspace Analysis of Generated Code Works”.

Click ✚ to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

## Tips

- If you specify a function for the option `Initialization functions (-functions-called-before-loop)`, you cannot specify it for this option.

## Command-Line Information

**Parameter:** `-functions-called-after-loop`

**No Default**

**Value:** `*function1*[,*function2*[,...]]`

**Example (Bug Finder):** `polyspace-bug-finder -sources *file_name* -main-generator -functions-called-after-loop myfunc`

**Example (Code Prover):** `polyspace-code-prover -sources *file_name* -main-generator -functions-called-after-loop myfunc`

**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-main-generator -functions-called-after-loop myfunc`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -functions-called-after-loop myfunc`

## See Also

`Verify model generated code (-main-generator)` | `Initialization functions (-functions-called-before-loop)` | `Step functions (-functions-called-in-loop)`

### Topics

“Configure Polyspace Options in Simulink”
“How Polyspace Analysis of Generated Code Works”

# Parameters (`-variables-written-before-loop`)

Specify variables that the generated `main` must initialize before the cyclic code loop

## Description

*This option is automatically set if you run Polyspace from Simulink or MATLAB on generated code. If you run Polyspace on generated code outside Simulink or MATLAB, set this option manually.*

Specify variables that the generated `main` must initialize before the cyclic code loop begins. Before the loop begins, Polyspace considers these variables to have any value allowed by their type.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node. You see this option only if you open a project configuration from Simulink.

**Command line and options file**: Use the option `-variables-written-before-loop`. See “Command-Line Information” on page 2-244.

## Settings

**Default:** `none`

`none`

The generated `main` does not initialize variables.

`all`

The generated `main` initializes all variables except those declared with keyword `const`.

`custom`

The generated `main` only initializes variables that you specify. Click ✚ to add a field. Enter variable name. For C++ class members, use the syntax `className::variableName`.

## Command-Line Information

**Parameter:** `-variables-written-before-loop`<br>
**Value:** `none` | `all` | `custom=`*`variable1`*`[,`*`variable2`*`[,...]]`<br>
**Default:** `none`<br>
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-main-generator -variables-written-before-loop all`<br>
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-main-generator -variables-written-before-loop all`<br>
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-main-generator -variables-written-before-loop all`<br>
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -variables-written-before-loop all`

## See Also

Verify model generated code (-main-generator) | Inputs (-variables-written-in-loop)

### Topics

“Configure Polyspace Options in Simulink”
“How Polyspace Analysis of Generated Code Works”

# Inputs (`-variables-written-in-loop`)

Specify variables that the generated `main` must initialize in the cyclic code loop

## Description

*This option is automatically set if you run Polyspace from Simulink or MATLAB on generated code. If you run Polyspace on generated code outside Simulink or MATLAB, set this option manually.*

Specify variables that the generated `main` must initialize at the beginning of every iteration of the cyclic code loop. At the beginning of every loop iteration, Polyspace considers these variables to have any value allowed by their type.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node. You see this option only if you open a project configuration from Simulink.

**Command line and options file**: Use the option `-variables-written-in-loop`. See “Command-Line Information” on page 2-246.

## Settings

**Default:** `none`

`none`

The generated `main` does not initialize variables.

`all`

The generated `main` initializes all variables except those declared with keyword `const`.

`custom`

The generated `main` only initializes variables that you specify. Click ✚ to add a field. Enter variable name. For C++ class members, use the syntax `className::variableName`.

## Command-Line Information

**Parameter:** `-variables-written-in-loop`
**Value:** `none` | `all` | `custom=`*`variable1`*`[,`*`variable2`*`[,...]]`
**Default:** `none`
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-main-generator -variables-written-in-loop all`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-main-generator -variables-written-in-loop all`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-main-generator -variables-written-in-loop all`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -variables-written-in-loop all`

## See Also

`Verify model generated code (-main-generator)` | `Parameters (-variables-written-before-loop)`

### Topics

"Configure Polyspace Options in Simulink"
"How Polyspace Analysis of Generated Code Works"

# Verify module or library (`-main-generator`)

Generate a `main` function if source files are modules or libraries that do not contain a `main`

## Description

*This option affects a Code Prover analysis only.*

Specify that Polyspace must generate a `main` function if it does not find one in the source files.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node.

**Command line and options file**: Use the option `-main-generator`. See “Command-Line Information” on page 2-249.

For the analogous option for model generated code, see `Verify model generated code (-main-generator)`.

### Why Use This Option

Use this option if you are verifying a module or library. A Code Prover analysis requires a `main` function. When verifying a module or library, your code might not have a `main`.

When you use this option, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

## Settings

● On (default)

Polyspace generates a `main` function if it does not find one in the source files. The generated `main`:

1. Initializes variables specified by `Variables to initialize (-main-generator-writes-variables)`.
2. Before calling other functions, calls the functions specified by `Initialization functions (-functions-called-before-main)`.
3. In all possible orders, calls the functions specified by `Functions to call (-main-generator-calls)`.
4. (C++ only) Calls class methods specified by `Class (-class-analyzer)` and `Functions to call within the specified classes (-class-analyzer-calls)`.

If you do not specify the function and variable options above, the generated `main`:

- Initializes all global variables except those declared with keywords `const` and `static`.
- In all possible orders, calls all functions that are not called anywhere in the source files. Polyspace considers that global variables can be written between two consecutive function

calls. Therefore, in each called function, global variables initially have the full range of values allowed by their type.

○ Off

Polyspace stops if a main function is not present in the source files.

## Tips

- If a main function is present in your source files, the verification uses that main function, irrespective of whether you enable or disable this option.

  The option is relevant only if a main function is not present in your source files.
- If you use the option Verify whole application (default on the command line), your code must contain a main function. Otherwise you see the error:

  ```
  Error: required main procedure not found
  ```

  If your code does not contain a main function, use this option to generate a main function.
- If you specify multitasking options, the verification ignores your specifications for main generation. Instead, the verification introduces an empty main function.

  For more information on the multitasking options, see “Configuring Polyspace Multitasking Analysis Manually”.

## Command-Line Information

**Parameter:** -main-generator
**Default:** Off
**Example (Code Prover):** polyspace-code-prover -sources *file_name* -main-generator
**Example (Code Prover Server):** polyspace-code-prover-server -sources *file_name* -main-generator

## See Also

Verify whole application | Variables to initialize (-main-generator-writes-variables) | Initialization functions (-functions-called-before-main) | Functions to call (-main-generator-calls) | Class (-class-analyzer) | Functions to call within the specified classes (-class-analyzer-calls)

### Topics

“Specify Polyspace Analysis Options”
“Verify C Application Without main Function” (Polyspace Code Prover)
“Verify C++ Classes” (Polyspace Code Prover)

# Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)

Specify that environment pointers can be unsafe to dereference unless constrained otherwise

## Description

*This option affects a Code Prover analysis only.*

*This option is not available for code generated from MATLAB code or Simulink models.*

Specify that the verification must consider environment pointers as unsafe unless otherwise constrained. Environment pointers are pointers that can be assigned values outside your code.

Environment pointers include:

- Global or `extern` pointers.
- Pointers returned from stubbed functions.

  A function is stubbed if your code does not contain the function definition or you override a function definition by using the option `Functions to stub (-functions-to-stub)`.
- Pointer parameters of functions whose calls are generated by the software.

  A function call is generated if you verify a module or library and the module or library does not have an explicit call to the function. You can also force a function call to be generated with the option `Functions to call (-main-generator-calls)`.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

**Command line and options file**: Use the option `-stubbed-pointers-are-unsafe`. See “Command-Line Information” on page 2-252.

### Why Use This Option

Use this option so that the verification makes more conservative assumptions about pointers from external sources.

If you specify this option, the verification considers that environment pointers can have a `NULL` value. If you read an environment pointer without checking for `NULL`, the **Illegally dereferenced pointer** check shows a potential error in orange. The message associated with the orange check shows the pointer can be `NULL`.

## Settings

☑ On

The verification considers that environment pointers can have a `NULL` value.

☐ Off (default)

The verification considers that environment pointers:

- Cannot have a `NULL` value.
- Points within allowed bounds.

## Tips

- Enable this option during the integration phase. In this phase, you provide complete code for verification. Even if an orange check originates from external sources, you are likely to place protections against unsafe pointers from such sources. For instance, if you obtain a pointer from an unknown source, you check the pointer for `NULL` value.

  Disable this option during the unit testing phase. In this phase, you focus on errors originating from your unit.
- If you are verifying code implementation of AUTOSAR runnables, Code Prover assumes that pointer arguments to runnables and pointers returned from `Rte_` functions are not `NULL`. You cannot use this option to change the assumption. See “Run Polyspace on AUTOSAR Code with Conservative Assumptions” (Polyspace Code Prover).
- If you enable this option, the number of orange checks in your code might increase.

| Environment Pointers Safe | Environment Pointers Unsafe |
|---|---|
| The **Illegally dereferenced pointer** check is green. The verification assumes that `env_ptr` is not NULL and any dereference is within allowed bounds. The verification assumes that the result of the dereference is full range. For instance, in this case, the return value has the full range of type `int`.<br>`int func (int *env_ptr) {`<br>`    return *env_ptr;`<br>`}` | The **Illegally dereferenced pointer** check is orange. The verification assumes that `env_ptr` can be NULL.<br>`int func (int *env_ptr) {`<br>`    return *env_ptr;`<br>`}` |

  If you enable this option, the number of gray checks might decrease.

| Environment Pointers Safe | Environment Pointers Unsafe |
|---|---|
| The verification assumes that `env_ptr` is not NULL. The `if` condition is always true and the `else` block is unreachable.<br>`#include <stdlib.h>`<br>`int func (int *env_ptr) {`<br>`    if(env_ptr!=NULL)`<br>`            return *env_ptr;`<br>`    else`<br>`            return 0;`<br>`}` | The verification assumes that `env_ptr` can be NULL. The `if` condition is not always true and the `else` block can be reachable.<br>`#include <stdlib.h>`<br>`int func (int *env_ptr) {`<br>`    if(env_ptr!=NULL)`<br>`            return *env_ptr;`<br>`    else`<br>`            return 0;`<br>`}` |

- Instead of considering all environment pointers as safe or unsafe, you can individually constrain some of the environment pointers. See the description of **Initialize Pointer** in “External Constraints for Polyspace Analysis”.

When you individually constrain a pointer, you first specify an **Init Mode**, and then specify through the **Initialize Pointer** option whether the pointer is `Null`, `Not Null`, or `Maybe Null`. Depending on the **Init Mode**, you can either override the global specification for all environment pointers or not.

- If you set the **Init Mode** of the pointer to `INIT` or `PERMANENT`, your selection for **Initialize Pointer** overrides your specification for this option. For instance, if you specify `Not NULL` for an environment pointer `ptr`, the verification assumes that `ptr` is not NULL even if you specify that environment pointers must be considered unsafe.
- If you set the **Init Mode** to `MAIN GENERATOR`, the verification uses your specification for this option.

  For pointers returned from stubbed functions, the option `MAIN GENERATOR` is not available. If you override the global specification for such a pointer through the **Initialize Pointer** option in constraints, you cannot toggle back to the global specification without changing the **Initialize Pointer** option too.

- If you disable this option, the verification considers that dereferences at all pointer depths are valid.

  For instance, all the dereferences are considered valid in this code:

```
int*** stub(void);

void func2() {
        int ***ptr = stub();
        int **ptr2 = *ptr;
        int *ptr3 = *ptr2;
}
```

## Command-Line Information

**Parameter:** `-stubbed-pointers-are-unsafe`
**Default**: Off
**Example (Code Prover)**: `polyspace-code-prover -sources` *`file_name`* `-stubbed-pointers-are-unsafe`
**Example (Code Prover Server)**: `polyspace-code-prover-server -sources` *`file_name`* `-stubbed-pointers-are-unsafe`

# Version History

**Introduced in R2016b**

## See Also

`Constraint setup (-data-range-specifications)`

### Topics

"Specify Polyspace Analysis Options"
"Specify External Constraints for Polyspace Analysis"
"External Constraints for Polyspace Analysis"

# Consider volatile qualifier on fields (-consider-volatile-qualifier-on-fields)

Assume that `volatile` qualified structure fields can have all possible values at any point in code

## Description

*This option affects a Code Prover analysis only.*

Specify that the verification must take into account the `volatile` qualifier on fields of a structure.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

**Command line and options file**: Use the option `-consider-volatile-qualifier-on-fields`. See "Command-Line Information" on page 2-255.

### Why Use This Option

The `volatile` qualifier on a variable indicates that the variable value can change between successive operations even if you do not explicitly change it in your code. For instance, if `var` is a `volatile` variable, the consecutive operations `res = var; res =var;` can result in two different values of `var` being read into `res`.

Use this option so that the verification emulates the `volatile` qualifier for structure fields. If you select this option, the software assumes that a `volatile` structure field has a full range of values at any point in the code. The range is determined only by the data type of the structure field.

## Settings

☑ On

The verification considers the `volatile` qualifier on fields of a structure.

In the following example, the verification considers that the field `val1` can have all values allowed for the `int` type at any point in the code.

```
struct myStruct {
   volatile int val1;
   int val2;
};
```

Even if you write a specific value to `val1` and read the variable in the next operation, the variable read results in any possible value.

```
struct myStruct myStructInstance;
myStructInstance.val1 = 1;
assert (myStructInstance.val1 == 1); // Assertion can fail
```

□ Off (default)

The verification ignores the `volatile` qualifier on fields of a structure.

In the following example, the verification ignores the qualifier on field `val1`.

```
struct myStruct {
   volatile int val1;
   int val2;
};
```

If you write a specific value to `val1` and read the variable in the next operation, the variable read results in that specific value.

```
struct myStruct myStructInstance;
myStructInstance.val1 = 1;
assert (myStructInstance.val1 == 1); // Assertion passes
```

## Tips

- If your volatile fields do not represent values read from hardware and you do not expect their values to change between successive operations, disable this option. You are using the `volatile` qualifier for some other reason and the verification does not need to consider full range for the field values.
- If you enable this option, the number of red, gray, and green checks in your code can decrease. The number of orange checks can increase.

  In the following example, a red or green check changes to orange or a gray check goes away when the option is used. Considering the `volatile` qualifier changes the check color. These examples use the following structure definition:

```
struct myStruct {
   volatile int field1;
   int field2;
};
```

| Color Without Option | Result Without Option | Result With Option |
|---|---|---|
| Green | `void main(){`<br>`   struct myStruct structVal;`<br>`   structVal.field1 = 1;`<br>`   assert(structVal.field1 == 1);`<br>`}` | `void main(){`<br>`   struct myStruct structVal;`<br>`   structVal.field1 = 1;`<br>`   assert(structVal.field1 ==1);`<br>`}` |
| Red | `void main(){`<br>`   struct myStruct structVal;`<br>`   structVal.field1 = 1;`<br>`   assert(structVal.field1 != 1);`<br>`}` | `void main(){`<br>`   struct myStruct structVal;`<br>`   structVal.field1 = 1;`<br>`   assert(structVal.field1 !=1);`<br>`}` |

| Color Without Option | Result Without Option | Result With Option |
|---|---|---|
| Gray | `void main(){`<br>`   struct myStruct structVal;`<br>`   structVal.field1 = 1;`<br>`   if (structVal.field1 != 1)`<br>`   {`<br>`   /* Perform operation */`<br>`   }`<br>`}` | `void main(){`<br>`   struct myStruct structVal;`<br>`   structVal.field1 = 1;`<br>`   if (structVal.field1 != 1)`<br>`   {`<br>`   /* Perform operation */`<br>`   }`<br>`}` |

- In C++ code, the option also applies to class members.

## Command-Line Information

**Parameter:** `-consider-volatile-qualifier-on-fields`
**Default**: Off
**Example (Code Prover)**: `polyspace-code-prover -sources` *`file_name`* `-consider-volatile-qualifier-on-fields`
**Example (Code Prover Server)**: `polyspace-code-prover-server -sources` *`file_name`* `-consider-volatile-qualifier-on-fields`

# Version History

**Introduced in R2016b**

## See Also

### Topics

"Specify Polyspace Analysis Options"

# Float rounding mode (`-float-rounding-mode`)

Specify rounding modes to consider when determining the results of floating point arithmetic

## Description

*This option affects a Code Prover analysis only.*

Specify the rounding modes to consider when determining the results of floating-point arithmetic.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Verification Assumptions** node.

**Command line and options file**: Use the option `-float-rounding-mode`. See “Command-Line Information” on page 2-258.

### Why Use This Option

The default verification uses the round-to-nearest mode.

Use the rounding mode `all` if your code contains routines such as `fesetround` to specify a rounding mode other than round-to-nearest. Although the verification ignores the `fesetround` specification, it considers all rounding modes including the rounding mode that you specified. Alternatively, for targets that can use extended precision (for instance, using the flag `-mfpmath=387`), use the rounding mode `all`. However, for your Polyspace analysis results to agree with run-time behavior, you must prevent use of extended precision through a flag such as `-ffloat-store`.

Otherwise, continue to use the default rounding mode `to-nearest`. Because all rounding modes are considered when you specify `all`, you can have many orange **Overflow** checks resulting from overapproximation.

## Settings

**Default:** `to-nearest`

`to-nearest`

The verification assumes the round-to-nearest mode.

`all`

The verification assumes all rounding modes for each operation involving floating-point variables. The following rounding modes are considered: round-to-nearest, round-towards-zero, round-towards-positive-infinity, and round-towards-negative-infinity.

## Tips

- The Polyspace analysis uses floating-point arithmetic that conforms to the IEEE® 754 standard. For instance, the arithmetic uses floating point instructions present in the SSE instruction set. The GNU C flag `-mfpmath=sse` enforces use of this instruction set. If you use the GNU C compiler with this flag to compile your code, your Polyspace analysis results agree with your run-time behavior.

However, if your code uses extended precision, for instance using the GNU C flag `-mfpmath=387`, your Polyspace analysis results might not agree with your run-time behavior in some corner cases. See some examples of these corner cases in `codeprover_limitations.pdf` in *`polyspaceroot`*`\polyspace\verifier\code_prover_desktop`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

To prevent use of extended precision, on targets without SSE support, you can use a flag such as `-ffloat-store`. For your Polyspace analysis, use `all` for rounding mode to account for double rounding.

- The **Overflow** check uses the rounding modes that you specify. For instance, the following table shows the difference in the result of the check when you change your rounding modes.

<table>
<tr><th>Rounding mode: <code>to-nearest</code></th><th>Rounding mode: <code>all</code></th></tr>
<tr>
<td>
If results of floating-point operations are rounded to nearest values:
<ul>
<li>In the first addition operation, <code>eps1</code> is just large enough that the value nearest to <code>FLT_MAX + eps1</code> is greater than <code>FLT_MAX</code>. The <b>Overflow</b> check is red.</li>
<li>In the second addition operation, <code>eps2</code> is just small enough that the value nearest to <code>FLT_MAX + eps2</code> is <code>FLT_MAX</code>. The <b>Overflow</b> check is green.</li>
</ul>
<pre>
#include &lt;float.h&gt;
#define eps1 0x1p103
#define eps2 0x0.FFFFFFp103

float func(int ch) {
    float left_op = FLT_MAX;
    float right_op_1 = eps1, \
right_op_2 = eps2;
    switch(ch) {
    case 1:
        return (left_op +\
right_op_1);
    case 2:
        return (left_op +\
right_op_2);
    default:
        return 0;
    }
}
</pre>
</td>
<td>
Besides to-nearest mode, the <b>Overflow</b> check also considers other rounding modes.
<ul>
<li>In the first addition operation, in to-nearest mode, the value nearest to <code>FLT_MAX + eps1</code> is greater than <code>FLT_MAX</code>, so the addition overflows. But if rounded towards negative infinity, the result is <code>FLT_MAX</code>, so the addition does not overflow. Combining these two rounding modes, the <b>Overflow</b> check is orange.</li>
<li>In the second addition operation, in to-nearest mode, the value nearest to <code>FLT_MAX + eps2</code> is <code>FLT_MAX</code>, so the addition does not overflow. But if rounded towards positive infinity, the result is greater than <code>FLT_MAX</code>, so the addition overflows. Combining these two rounding modes, the <b>Overflow</b> check is orange.</li>
</ul>
<pre>
#include &lt;float.h&gt;
#define eps1 0x1p103
#define eps2 0x0.FFFFFFp103

float func(int ch) {
    float left_op = FLT_MAX;
     float right_op_1 = eps1, \
 right_op_2 = eps2;
    switch(ch) {
    case 1:
        return (left_op +\
right_op_1);
    case 2:
        return (left_op +\
right_op_2);
    default:
        return 0;
    }
}
</pre>
</td>
</tr>
</table>

If you set the rounding mode to `all` and obtain an orange **Overflow** check, to determine how the overflow can occur, consider all rounding modes.

## Command-Line Information

**Parameter:** `-float-rounding-mode`
**Value:** `to-nearest` | `all`
**Default:** `to-nearest`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-float-rounding-mode all`

**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-float-rounding-mode all`

## Version History

**Introduced in R2016a**

## See Also

`Overflow`

### Topics

"Specify Polyspace Analysis Options"

# Ignore assembly code (`-ignore-assembly-code`)

Specify that assembly instructions in C/C++ code cannot modify C/C++ variables

## Description

*This option affects a Code Prover analysis only.*

Enable an analysis mode that ignores assembly instructions in C/C++ code. In this mode, the analysis assumes that assembly instructions in a C/C++ function cannot modify local variables of the function.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **analysis Assumptions** node.

**Command line and options file**: Use the option `-ignore-assembly-code`. See “Command-Line Information” on page 2-261.

### Why Use This Option

Use this option to reduce orange checks from overapproximations around assembly instructions in C/C++ code.

By default, Code Prover makes the conservative assumption that following assembly instructions in a function, local variables can take any value allowed by their data types. This assumption can lead to many orange checks following assembly instructions. If you use assembly instructions primarily for no-ops such as introducing delays or for loading global variables, you can enable an analysis mode that ignores the assembly instructions.

## Settings

☑ On

The analysis considers that assembly instructions in a function cannot modify local variables of the function. Use this option if assembly instructions in your project do not modify local variables of a function.

Consider these examples:

- The analysis assumes that the assembly instructions cannot modify the local variable `val1`. Therefore, the results show a red **Non-initialized variable** check when `val1` is returned.

  ```
  int func(void) {
      int val1;
      asm("NOP");
      //Instructions that do not modify val1
      //...
      return val1;
  }
  ```

- The analysis assumes that the assembly instructions cannot modify the local variables `val1` and `val2`. Therefore, the results show a green **Overflow** check when the two variables are added.

  ```
  int func(void) {
      int val1 = 0, val2 = 0;
      asm("NOP");
      //Instructions that do not modify val1 and val2
      //...
      return (val1+val2);
  }
  ```

□ Off (default)

The analysis considers that assembly instructions in a function can modify local variables of the function. Use this option if assembly instructions in your project can modify local variables of a function.

Consider these examples:

- The analysis assumes that the assembly instructions can potentially modify the local variable `val1`. Therefore, the results show an orange **Non-initialized variable** check when `val1` is returned.

  ```
  int func(void) {
      int val1;
      asm("mov 4%0,%%eax"::"m"(val1));
      //Instructions that do not modify val1
      //...
      return val1;
  }
  ```
- The analysis assumes that the assembly instructions can modify the local variables `val1` and `val2` (and write any possible value allowed by their data types). Therefore, the results show an orange **Overflow** check when the two variables are added.

  ```
  int func(void) {
      int val1 = 0, val2 = 0;
      asm("mov 4%0,%%eax"::"m"(val1));
      //Instructions that do not modify val1 and val2
      //...
      return (val1+val2);
  }
  ```

## Command-Line Information

**Parameter:** `-ignore-assembly-code`
**Default**: Off
**Example (Code Prover)**: `polyspace-code-prover -sources` *`file_name`* `-ignore-assembly-code`
**Example (Code Prover Server)**: `polyspace-code-prover-server -sources` *`file_name`* `-ignore-assembly-code`

# Version History

**Introduced in R2023a**

## See Also

"Code Prover Assumptions About Assembly Code" (Polyspace Code Prover)

### Topics

"Specify Polyspace Analysis Options"
"Specify External Constraints for Polyspace Analysis"
"External Constraints for Polyspace Analysis"

# Allow negative operand for left shifts (-allow-negative-operand-in-shift)

Allow left shift operations on a negative number

## Description

*This option affects a Code Prover analysis only.*

Specify that the verification must allow left shift operations on a negative number.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

**Command line and options file**: Use the option `-allow-negative-operand-in-shift`. See “Command-Line Information” on page 2-263.

### Why Use This Option

According to the C99 standard (sec 6.5.7), the result of a left shift operation on a negative number is undefined. Following the standard, the verification produces a red check on left shifts of negative numbers.

If your compiler has a well-defined behavior for left shifts of negative numbers, set this option. Note that allowing left shifts of negative numbers can reduce the cross-compiler portability of your code.

## Settings

☑ On

The verification allows shift operations on a negative number, for instance, `-2 << 2`.

☐ Off (default)

If a shift operation is performed on a negative number, the verification generates an error.

## Command-Line Information

**Parameter:** `-allow-negative-operand-in-shift`
**Default**: Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-allow-negative-operand-in-shift`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-allow-negative-operand-in-shift`

## See Also

`Invalid shift operations`

**Topics**
“Specify Polyspace Analysis Options”
“Modify or Disable Code Prover Run-Time Checks” (Polyspace Code Prover)

# Consider non finite floats (-allow-non-finite-floats)

Enable an analysis mode that incorporates infinities and NaNs

## Description

Enable an analysis mode that incorporates infinities and NaNs for floating point operations.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

**Command line and options file**: Use the option `-allow-non-finite-floats`. See “Command-Line Information” on page 2-267.

### Why Use This Option

#### Code Prover

By default, the analysis does not incorporate infinities and NaNs. For instance, the analysis terminates the execution thread where a division by zero occurs and does not consider that the result could be infinite.

If you use functions such as `isinf` or `isnan` and account for infinities and NaNs in your code, set this option. When you set this option and a division by zero occurs for instance, the execution thread continues with infinity as the result of the division.

Set this option alone if you are sure that you have accounted for infinities and NaNs in your code. Using the option alone effectively disables many numerical checks on floating point operations. If you have generally accounted for infinities and NaNs, but you are not sure that you have considered all situations, set these additional options:

- `Infinities (-check-infinite)`: Use `warn-first`.
- `NaNs (-check-nan)`: Use `warn-first`.

#### Bug Finder

If the analysis flags comparisons using `isinf` or `isnan` as dead code, use this option. By default, a Bug Finder analysis does not incorporate infinities and NaNs.

## Settings

☑ On

The analysis allows infinities and NaNs. For instance, in this mode:

- The analysis assumes that floating-point operations can produce results such as infinities and NaNs.

  By using options `Infinities (-check-infinite)` and `NaNs (-check-nan)`, you can choose to highlight operations that produce nonfinite results and stop the execution threads where the nonfinite results occur. These options are not available for a Bug Finder analysis.

- The analysis assumes that floating-point variables with unknown values can have any value allowed by their type, including infinite or NaN. Floating-point variables with unknown values include volatile variables and return values of stubbed functions.

□ Off (default)

The analysis does not allow infinities and NaNs. For instance, in this mode:

- The Code Prover analysis produces a red check on a floating-point operation that produces an infinity or a NaN as the only possible result on all execution paths. The verification produces an orange check on a floating-point operation that can potentially produce an infinity or NaN.
- The Code Prover analysis assumes that floating-point variables with unknown values are full-range but finite.
- The Bug Finder analysis shows comparisons with infinity using `isinf` as dead code.

## Tips

- The IEEE 754 Standard allows special quantities such as infinities and `NaN` so that you can handle certain numerical exceptions without aborting the code. Some implementations of the C standard support infinities and `NaN`.

  - If your compiler supports infinities and `NaN`s and you account for them explicitly in your code, use this option so that the verification also allows them.

    For instance, if a division results in infinity, in your code, you specify an alternative action. Therefore, you do not want the verification to highlight division operations that result in infinity.
  - If your compiler supports infinities and `NaN`s but you are not sure if you account for them explicitly in your code, use this option so that the verification incorporates infinities and `NaN`s. Use the options `-check-nan` and `-check-infinite` with argument `warn` so that the verification highlights operations that result in infinities and `NaN`s, but does not stop the execution thread. These options are not available for a Bug Finder analysis.
- If you run a Code Prover analysis and use this option, checkers for overflow, division by zero and other numerical run-time errors are disabled. See “Numerical Checks” (Polyspace Code Prover).

  If you run a Bug Finder analysis and use this option:

  - These checkers are disabled:

    - Bug Finder defects: `Float conversion overflow`, `Float division by zero`, `Invalid use of standard library floating point routine`, `Float overflow`.
    - CERT C rules and recommendation: ,`CERT C: Rule FLP34-C`, `CERT C: Rule FLP32-C`, `CERT C: Rec. FLP03-C`, `CERT C: Rec. FLP06-C`.
    - CERT C++ rules: `CERT C++: FLP34-C`, `CERT C++: FLP32-C`.
    - AUTOSAR C++14 rule: `AUTOSAR C++14 Rule A0-4-4`.
    - MISRA C:2004 rule: MISRA C:2004 rule 20.3.
  - These checker might show less violations: `MISRA C:2012 Dir 4.1`

  - These checkers might show false positives: `Floating point comparison with equality operators`, `AUTOSAR C++14 Rule M6-2-2`, `MISRA C:2012 Dir 1.1`.

- If you select this option, the number and type of Code Prover checks in your code can change.

  For instance, in the following example, when you select the option, the results have one less red check and three more green checks.

<table>
<tr><th>Infinities and NaNs Not Allowed</th><th>Infinities and NaNs Allowed</th></tr>
<tr><td>Code Prover produces a <b>Division by zero</b> error and stops verification.
<pre>
double func(void) {
    double x=1.0/0.0;
    double y=1.0/x;
    double z=x-x;
    return z;
}
</pre></td><td>If you select this option, Code Prover does not check for a <b>Division by zero</b> error.
<pre>
double func(void) {
    double x=1.0/0.0;
    double y=1.0/x;
    double z=x-x;
    return z;
}
</pre>
The analysis assumes that dividing by zero results in:<br>• Value of <code>x</code> equal to <code>Inf</code><br>• Value of <code>y</code> equal to 0.0<br>• Value of <code>z</code> equal to <code>NaN</code><br>In your analysis results in the Polyspace user interface, if you place your cursor on <code>y</code> and <code>z</code>, you can see the nonfinite values <code>Inf</code> and <code>NaN</code> respectively in the tooltip.</td></tr>
</table>

## Command-Line Information

**Parameter:** `-allow-non-finite-floats`
**Default**: Off
**Example (Bug Finder):** `polyspace-bug-finder -sources *file_name* -allow-non-finite-floats`
**Example (Code Prover):** `polyspace-code-prover -sources *file_name* -allow-non-finite-floats`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources *file_name* -allow-non-finite-floats`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources *file_name* -allow-non-finite-floats`

# Version History

**Introduced in R2016a**

## See Also

`Infinities (-check-infinite)` | `NaNs (-check-nan)` | “Numerical Defects” | “Numerical Checks” (Polyspace Code Prover)

### Topics

"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks" (Polyspace Code Prover)
"Modify Default Behavior of Bug Finder Checkers"

# Infinities (-check-infinite)

Specify how to handle floating-point operations that result in infinity

## Description

*This option affects a Code Prover analysis only.*

Specify how the analysis must handle floating-point operations that result in infinities.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See “Dependencies” on page 2-270 for other options you must also enable.

**Command line and options file**: Use the option `-check-infinite`. See “Command-Line Information” on page 2-270.

### Why Use This Option

Use this option to enable detection of floating-point operations that result in infinities.

If you specify that the analysis must consider nonfinite floats, by default, the analysis does not flag these operations. Use this option to detect these operations while still incorporating nonfinite floats.

## Settings

**Default:** `allow`

`allow`

The verification does not produce a check on the operation.

For instance, in the following code, there is no **Overflow** check.

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

`warn-first`

The verification produces a check on the operation. The check determines if the result of the operation is infinite when the operands themselves are not infinite. The verification does not terminate the execution thread that produces infinity.

If the verification detects an operation that produces infinity as the only possible result on all execution paths and the operands themselves are never infinite, the check is red. If the operation can potentially result in infinity, the check is orange.

For instance, in the following code, there is a nonblocking **Overflow** check for infinity.

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

Even though the **Overflow** check on the / operation is red, the verification continues. For instance, a green **Non-initialized local variable** check appears on x in the return statement.

forbid

The verification produces a check on the operation and terminates the execution thread that produces infinity.

If the check is red, the verification does not continue for the remaining code in the same scope as the check. If the check is orange, the verification continues but removes from consideration the variable values that produced infinity.

For instance, in the following code, there is a blocking **Overflow** check for infinity.

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

The verification stops because the **Overflow** check on the / operation is red. For instance, a **Non-initialized local variable** check does not appear on x in the return statement.

### Dependencies

To use this option, you must enable the verification mode that incorporates infinities and NaNs. See Consider non finite floats (-allow-non-finite-floats).

### Command-Line Information

**Parameter:** -check-infinite
**Value:** allow | warn-first | forbid
**Default:** allow
**Example (Code Prover):** polyspace-code-prover -sources *file_name* -check-infinite forbid
**Example (Code Prover Server):** polyspace-code-prover-server -sources *file_name* -check-infinite forbid

## Version History

**Introduced in R2016a**

### See Also

**Polyspace Analysis Options**
Consider non finite floats (-allow-non-finite-floats) | NaNs (-check-nan)

**Polyspace Results**
Overflow

**Topics**
“Specify Polyspace Analysis Options”

“Modify or Disable Code Prover Run-Time Checks” (Polyspace Code Prover)

# Check that global variables are initialized after warm reboot (`-check-globals-init`)

Check that global variables are assigned values in designed initialization code

## Description

*This option affects a Code Prover analysis only.*

Specify that Polyspace must check whether all non-const global variables (and local static variables) are explicitly initialized at declaration or within a section of code marked as initialization code.

To indicate the end of initialization code, you enter the line

```
#pragma polyspace_end_of_init
```

in the `main` function (only once). The initialization code starts from the beginning of `main` and continues up to this pragma.

Since compilers ignore unrecognized pragmas, the presence of this pragma does not affect program execution.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

**Command line and options file**: Use the option `-check-globals-init`. See “Command-Line Information” on page 2-275.

### Why Use This Option

In a warm reboot, to save time, the bss segment of a program, which might hold variable values from a previous state, is not loaded. Instead, the program is supposed to explicitly initialize all non-const variables without default values before execution. You can use this option to delimit the initialization code and verify that all non-const global variables are indeed initialized in a warm reboot.

For instance, in this simple example, the global variable `aVar` is initialized in the initialization code section but the variable `anotherVar` is not.

```
int aVar;
const int aConst = -1;
int anotherVar;

int main() {
      aVar = aConst;
#pragma polyspace_end_of_init
      return 0;
}
```

## Settings

☑ On

Polyspace checks whether all global variables are initialized in the designated initialization code. The initialization code starts from the beginning from `main` and continues up to the pragma `polyspace_end_of_init`.

The results are reported using the check `Global variable not assigned a value in initialization code`.

○ Off (default)

Polyspace does not check for initialization of global variables in a designated code section.

However, the verification continues to check if a variable is initialized at the time of use. The results are reported using the check `Non-initialized variable`.

## Dependencies

You can use this option and designate a section of code as initialization code only if:

- Your program contains a `main` function and you use the option `Verify whole application` (implicitly set by default at command line).
- You set `Source code language (-lang)` to `C`.

Note that the pragma must appear only once in the `main` function. The pragma can appear before or after variable declarations but must appear after type definitions (`typedef`-s).

You cannot use this option with the following options:

- `Disable checks for non-initialization (-disable-initialization-checks)`
- `Verify files independently (-unit-by-unit)`
- `Show global variable sharing and usage only (-shared-variables-mode)`

## Tips

- You can use this option along with the option `Verify initialization section of code only (-init-only-mode)` to check the initialization code before checking the remaining program.

  This approach has the following benefits compared to checking the entire code in one run:

  - Run-time errors in the initialization code can invalidate analysis of the remaining code. You can run a comparatively quicker check on the initialization code before checking the remaining program.
  - You can review results of the checker `Global variable not assigned a value in initialization code` relatively easily.

    Consider this example. There is an orange check on `var` because `var` might remain uninitialized when the `if` and `else if` statements are skipped.

    ```
    int var;
    ```

```
int checkSomething(void);
int checkSomethingElse(void);

int main() {
    int local_var;
    if(checkSomething())
    {
        var=0;
    }
    else if(checkSomethingElse()) {
        var=1;
    }
    #pragma polyspace_end_of_init
    var=2;
    local_var = var;
    return 0;
}
```

To review this check and understand when x might be non-initialized, you have to browse through all instances of x on the **Variable Access** pane. If you check the initialization code alone, only the code in bold gets checked and you have to browse through only the instances in the initialization code.

- The check is only as good as your placement of the pragma polyspace_end_of_init. For instance:

  - Place the pragma only after initialization code ends.

    Otherwise, a variable might appear falsely uninitialized.
  - Try to place the pragma directly in the main function, that is, outside a block. If you place the pragma in a block, the check considers only those paths that end in the block.

    All paths that end in the block might have a variable initialized but paths that skip the block might let the variable go uninitialized. If you do place the pragma in a block, make sure that it is okay if a variable stays uninitialized outside the block.

    For instance, in this example, the variable var is initialized on all paths that end at the location of the pragma. The check is green despite the fact that the if block might be skipped, letting the variable go uninitialized.

```
int var;

int func();

int main() {
    int err = func();
    if(err) {
        var = 0;
 #pragma polyspace_end_of_init
    }
    int a = var;
    return 0;
}
```

The issue is detected by the checker if you place the pragma after the if block ends.

- Do not place the pragma in a loop.

  If you place the pragma in a loop, you can see results that are difficult to interpret. For instance, in this example, both `aVar` and `anotherVar` are initialized in one iteration of the loop. However, the pragma only considers the first iteration of the loop when it shows a green check for initialization. If a variable is initialized on a later iteration, the check is orange.

  ```
  int aVar;
  int anotherVar;

  void main() {
      for(int i=0; i<=1; i++) {
          if(i == 0)
              aVar = 0;
          else
              anotherVar = 0;
          #pragma polyspace_end_of_init
      }
  }
  ```

  The check is red if you verify initialization code alone and do not initialize a variable in the first loop iteration. To avoid these incorrect red or orange checks, do not place the pragma in a loop.
- To determine the initialization of a structure, a regular Code Prover analysis only considers fields that are used.

  If you check initialization code only using the option `Verify initialization section of code only (-init-only-mode)`, the analysis covers only a portion of the code and cannot determine if a variable is used beyond this portion. Therefore, the checks for initialization consider all structure fields, whether used or not.

## Command-Line Information

**Parameter:** `-check-globals-init`
**Default**: Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-check-globals-init`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-check-globals-init`

# Version History

**Introduced in R2020a**

## See Also

`Verify initialization section of code only (-init-only-mode)` | `Global variable not assigned a value in initialization code`

### Topics

“Specify Polyspace Analysis Options”
“Modify or Disable Code Prover Run-Time Checks” (Polyspace Code Prover)
“Code Prover Assumptions About Global Variable Initialization” (Polyspace Code Prover)

# NaNs (-check-nan)

Specify how to handle floating-point operations that result in NaN

## Description

*This option affects a Code Prover analysis only.*

Specify how the analysis must handle floating-point operations that result in NaN.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See “Dependencies” on page 2-277 for other options you must also enable.

**Command line and options file**: Use the option `-check-nan`. See “Command-Line Information” on page 2-277.

### Why Use This Option

Use this option to enable detection of floating-point operations that result in NaN-s.

If you specify that the analysis must consider nonfinite floats, by default, the analysis does not flag these operations. Use this option to detect these operations while still incorporating nonfinite floats.

## Settings

**Default:** `allow`

`allow`

The verification does not produce a check on the operation.

For instance, in the following code, there is no **Invalid operation on floats** check.

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

`warn-first`

The verification produces a check on the operation. The check determines if the result of the operation is NaN when the operands themselves are not NaN. For instance, the check flags the operation `val1 + val2` only if the result can be NaN when *both* `val1` and `val2` are not NaN. The verification does not terminate the execution thread that produces NaN.

If the verification detects an operation that produces NaN as the only possible result on all execution paths and the operands themselves are never NaN, the check is red. If the operation can potentially result in NaN, the check is orange.

For instance, in the following code, there is a nonblocking **Invalid operation on floats** check for NaN.

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

Even though the **Invalid operation on floats** check on the `-` operation is red, the verification continues. For instance, a green **Non-initialized local variable** check appears on `y` in the `return` statement.

`forbid`

The verification produces a check on the operation and terminates the execution thread that produces NaN.

If the check is red, the verification does not continue for the remaining code in the same scope as the check. If the check is orange, the verification continues but removes from consideration the variable values that produced a NaN.

For instance, in the following code, there is a blocking **Invalid operation on floats** check for NaN.

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

The verification stops because the **Invalid operation on floats** check on the `-` operation is red. For instance, a **Non-initialized local variable** check does not appear on `y` in the `return` statement.

The **Invalid operation on floats** check for NaN also appears on the `/` operation and is green.

## Dependencies

To use this option, you must enable the verification mode that incorporates infinities and NaNs. See `Consider non finite floats (-allow-non-finite-floats)`.

## Command-Line Information

**Parameter:** `-check-nan`
**Value:** `allow | warn-first | forbid`
**Default:** `allow`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-check-nan forbid`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-check-nan forbid`

# Version History

**Introduced in R2016a**

## See Also

**Polyspace Analysis Options**
`Consider non finite floats (-allow-non-finite-floats)` | `Infinities (-check-infinite)`

**Polyspace Results**
`Invalid operation on floats`

**Topics**
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks" (Polyspace Code Prover)

# Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)

Allow arithmetic on pointer to a structure field so that it points to another field

## Description

*This option affects a Code Prover analysis only.*

Specify that a pointer assigned to a structure field can point outside its bounds as long as it points within the structure.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See “Dependency” on page 2-280 for other options you must also enable.

**Command line and options file**: Use the option `-allow-ptr-arith-on-struct`. See “Command-Line Information” on page 2-280.

### Why Use This Option

Use this option to relax the check for illegally dereferenced pointers. Once you assign a pointer to a structure field, you can perform pointer arithmetic and use the result to access another structure field.

## Settings

☑ On

A pointer assigned to a structure field can point outside the bounds imposed by the field as long as it points within the structure. For instance, in the following code, unless you use this option, the verification will produce a red `Illegally dereferenced pointer` check:

```
void main(void) {
struct S {char a; char b; int c;} x;
char *ptr = &x.b;
ptr ++;
*ptr = 1; // Red on the dereference, because ptr points outside x.b
}
```

☐ Off (default)

A pointer assigned to a structure field can point only within the bounds imposed by the field.

## Tips

- The verification does not allow a pointer with negative offset values. This behavior occurs irrespective of whether you choose the option **Enable pointer arithmetic across fields**.
- Using this option can slightly increase the number of orange checks. The option relaxes the constraint that a pointer to a structure field cannot point to other fields of the structure. In

exchange for relaxing this constraint, the verification loses precision on the boundary of fields within a structure and treats the structure as a whole. Pointer dereferences that were previously green can now turn orange.

Use this option if you follow a policy of reviewing red checks only and you need to work around red checks from pointer arithmetic within a structure.

- Before using this option, consider the costs of using pointer arithmetic across different fields of a structure.

  Unlike an array, members of a structure can have different data types. For efficient storage, structures use padding to accommodate this difference. When you increment a pointer pointing to a structure member, you might not point to the next member. When you dereference this pointer, you cannot rely on what you are reading or writing to.

## Dependency

This option is available only if you set `Source code language (-lang)` to `C`.

## Command-Line Information

**Parameter:** `-allow-ptr-arith-on-struct`
**Default**: Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-allow-ptr-arith-on-struct`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-allow-ptr-arith-on-struct`

## See Also

`Allow incomplete or partial allocation of structures (-size-in-bytes)` | `Illegally dereferenced pointer`

### Topics

"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks" (Polyspace Code Prover)

# Detect stack pointer dereference outside scope (`-detect-pointer-escape`)

Find cases where a function returns a pointer to one of its local variables

## Description

*This option affects a Code Prover analysis only.*

Specify that the verification must detect cases where you access a variable outside its scope via dangling pointers. Such an access can happen, for example, when a function returns a pointer to a local variable and you dereference the pointer outside the function. The dereference causes undefined behavior because the local variable that the pointer points to does not live outside the function.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

**Command line and options file**: Use the option `-detect-pointer-escape`. See “Command-Line Information” on page 2-282.

### Why Use This Option

Use this option to enable detection of pointer escape.

## Settings

☑ On

The **Illegally dereferenced pointer** check performs an additional task, besides its usual specifications. When you dereference a pointer, the check also determines if you are accessing a variable outside its scope through the pointer. The check is:

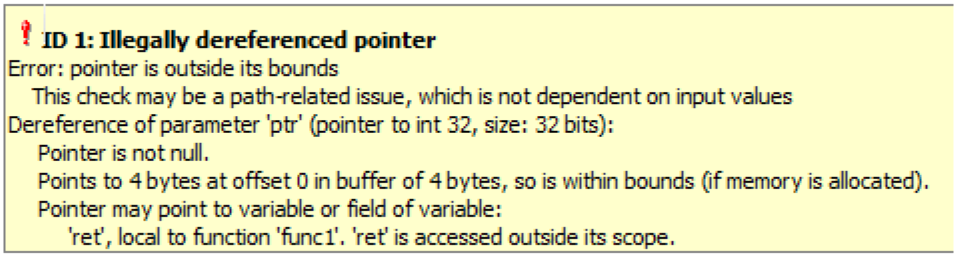
- Red, if all the variables that the pointer points to are accessed outside their scope.

  For instance, you dereference a pointer `ptr` in a function `func` that is called twice in your code. In both calls, when you perform the dereference `*ptr`, `ptr` is pointing to variables outside their scope. Therefore, the **Illegally dereferenced pointer** check is red.
- Orange, if only some of the variables that the pointer points to are accessed outside their scope.
- Green, if none of the variables that the pointer points to are accessed outside their scope, and other requirements of the check are also satisfied.

In the following code, if you enable this option, Polyspace Code Prover produces a red **Illegally dereferenced pointer** check on `*ptr`. Otherwise, the **Illegally dereferenced pointer** check on `*ptr` is green.

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

The **Result Details** pane displays a message indicating that `ret` is accessed outside its scope.



**ID 1: Illegally dereferenced pointer**

Error: pointer is outside its bounds

 This check may be a path-related issue, which is not dependent on input values

Dereference of parameter 'ptr' (pointer to int 32, size: 32 bits):

 Pointer is not null.

 Points to 4 bytes at offset 0 in buffer of 4 bytes, so is within bounds (if memory is allocated).

 Pointer may point to variable or field of variable:

  'ret', local to function 'func1'. 'ret' is accessed outside its scope.

□ Off (default)

When you dereference a pointer, the **Illegally dereferenced pointer** check does not check for whether you are accessing a variable outside its scope. The check is green even if the pointer dereference is outside the variable scope, as long as it satisfies these requirements:

- The pointer is not NULL.
- The pointer points within the memory buffer.

## Tips

The detection of stack pointer deference outside scope does not apply to certain types of pointers. For specific limitations, see “Limitations of Polyspace Verification” (Polyspace Code Prover).

## Command-Line Information

**Parameter**: `-detect-pointer-escape`
**Default**: Off
**Example (Code Prover)**: `polyspace-code-prover -sources` *`file_name`* `-detect-pointer-escape`
**Example (Code Prover Server)**: `polyspace-code-prover-server -sources` *`file_name`* `-detect-pointer-escape`

# Version History

**Introduced in R2015a**

## See Also

`Illegally dereferenced pointer`

### Topics

“Specify Polyspace Analysis Options”

"Modify or Disable Code Prover Run-Time Checks" (Polyspace Code Prover)

# Disable checks for non-initialization (`-disable-initialization-checks`)

Disable checks for non-initialized variables and pointers

## Description

*This option affects a Code Prover analysis only.*

Specify that Polyspace Code Prover must not check for non-initialization in your code.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

**Command line and options file**: Use the option `-disable-initialization-checks`. See "Command-Line Information" on page 2-285.

### Why Use This Option

Use this option if you do not want to detect instances of non-initialized variables.

## Settings

☑ On

Polyspace Code Prover does not perform the following checks:

- `Non-initialized local variable`: Local variable is not initialized before being read.
- `Non-initialized variable`: Variable other than local variable is not initialized before being read.
- `Non-initialized pointer`: Pointer is not initialized before being read.
- `Return value not initialized`: C function does not return value when expected.

Polyspace assumes that, at declaration:

- Variables have full-range of values allowed by their type.
- Pointers can be `NULL`-valued or point to a memory block at an unknown offset.

□ Off (default)

Polyspace Code Prover checks for non-initialization in your code. The software displays red checks if, for instance, a variable is not initialized and orange checks if a variable is initialized only on some execution paths.

## Tips

- If you select this option, the software does not report most violations of MISRA C:2004 rule 9.1, and `MISRA C:2012 Rule 9.1`.

- If you select this option, the number and type of orange checks in your code can change.

  For instance, the following table shows an additional orange check with the option enabled.

| Checks for Non-initialization Enabled | Checks for Non-initialization Disabled |
|---|---|
| `void func(int flag) {`<br>`    int var1,var2;`<br>`    if( flag==0) {`<br>`        var1=var2;`<br>`    }`<br>`    else {`<br>`        var1=0;`<br>`    }`<br>`    var2=var1 + 1;`<br>`}`<br>In this example, the software produces:<br>• A red **Non-initialized local variable** check on `var2` in the `if` branch. The verification continues as if only the `else` branch of the `if` statement exists.<br>• A green **Non-initialized local variable** check on `var1` in the last statement. `var1` has the assigned value 0.<br>• A green **Overflow** check on the `+` operation. | `void func(int flag) {`<br>`    int var1,var2;`<br>`    if( flag==0) {`<br>`        var1=var2;`<br>`    }`<br>`    else {`<br>`        var1=0;`<br>`    }`<br>`    var2=var1 + 1;`<br>`}`<br>In this example, the software:<br>• Does not produce **Non-initialized local variable** checks. At initialization, the software assumes that `var2` has full range of `int` values. Following the `if` statement, because the software considers both `if` branches, it assumes that `var1` also has full range of `int` values.<br>• Produces an orange **Overflow** check on the `+` operation. For instance, if `var1` has the maximum `int` value, adding 1 to it can cause an overflow. |

## Command-Line Information

**Parameter:** `-disable-initialization-checks`
**Default**: Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-disable-initialization-checks`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-disable-initialization-checks`

## See Also

### Topics

"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks" (Polyspace Code Prover)

# Permissive function pointer calls (`-permissive-function-pointer`)

Allow type mismatch between function pointers and the functions they point to

## Description

*This option affects a Code Prover analysis only.*

Specify that the verification must allow function pointer calls where the type of the function pointer does not match the type of the function.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See “Dependency” on page 2-288 for other options you must also enable.

**Command line and options file**: Use the option `-permissive-function-pointer`. See “Command-Line Information” on page 2-288.

### Why Use This Option

By default, Code Prover does not recognize calls through function pointers when a type mismatch occurs. Fix the type mismatch whenever possible.

Use this option if:

- You cannot fix the type mismatch, and
- The analysis does not cover a significant portion of your code because calls via function pointers are not recognized.

With sources that use function pointers extensively, enabling this option can cause loss in performance. This loss occurs because the verification has to consider more complex call graphs and more execution paths. In rare cases, the verification can run out of memory.

## Settings

☑ On

The verification must allow function pointer calls where the type of the function pointer does not match the type of the function. For instance, a function declared as `int f(int*)` can be called by a function pointer declared as `int (*fptr)(void*)`.

Only type mismatches between pointer types are allowed. Type mismatches between nonpointer types cause compilation errors. For instance, a function declared as `int f(int)` cannot be called by a function pointer declared as `int (*fptr)(double)`.

□ Off (default)

The verification must require that the argument and return types of a function pointer and the function it calls are identical.

Type mismatches are detected with the check Correctness condition.

## Tips

- Using this option can increase the number of orange checks. Some of these orange checks can reveal a real issue with the code.

  Consider these examples where a type mismatch occurs between the function pointer type and the function that it points to:

  - In this example, the function pointer obj_fptr has an argument that is a pointer to a three-element array. However, it points to a function whose corresponding argument is a pointer to a four-element array. In the body of foo, four array elements are read and incremented. The fourth element does not exist and the ++ operation reads a meaningless value.

```
typedef int array_three_elements[3];
typedef void (*fptr)(array_three_elements*);

typedef int array_four_elements[4];
void foo(array_four_elements*);

void main() {
 array_three_elements arr[3] = {0,0,0};
 array_three_elements *ptr;
 fptr obj_fptr;

 ptr = &arr;
 obj_fptr = &foo;

 //Call via function pointer
 obj_fptr(&ptr);
}

void foo(array_four_elements* x) {
    int i = 0;
    int *current_pos;

    for(i = 0; i< 4; i++) {
      current_pos = (*x) + i;
      (*current_pos)++;
    }
}
```

    Without this option, an orange Correctness condition check appears on the call obj_fptr(&ptr) and the function foo is not verified. If you use this option, the body of foo contains several orange checks. Review the checks carefully and make sure that the type mismatch does not cause issues.

  - In this example, the function pointer has an argument that is a pointer to a structure with three float members. However, the corresponding function argument is a pointer to an unrelated structure with one array member. In the function body, the strlen function is used assuming the array member. Instead the strlen call reads the float members and can read meaningless values, for instance, values stored in the structure padding.

```
#include <string.h>
struct point {
  float x;
  float y;
  float z;
};
struct message {
  char msg[10] ;
};
void foo(struct message*);

void main() {
  struct point pt = {3.14, 2048.0, -1.0} ;
  void (*obj_fptr)(struct point *) ;

  obj_fptr = &foo;

  //Call via function pointer
  obj_fptr(&pt);
}

void foo(struct message* x) {
  int y = strlen(x->msg) ;
}
```

Without this option, an orange check appears on the call `obj_fptr(&pt)` and the function `foo` is not verified. If you use this option, the function contains an orange check on the `strlen` call. Review the check carefully and make sure that the type mismatch does not cause issues.

## Dependency

This option is available only if you set `Source code language (-lang)` to `C`.

## Command-Line Information

**Parameter:** `-permissive-function-pointer`
**Default**: Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-lang c -permissive-function-pointer`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-lang c -permissive-function-pointer`

## See Also

`Correctness condition`

### Topics

"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks" (Polyspace Code Prover)

# Overflow mode for signed integer (`-signed-integer-overflows`)

Specify whether result of overflow is wrapped around or truncated

## Description

*This option affects a Code Prover analysis only.*

Specify whether Polyspace flags signed integer overflows and whether the analysis wraps the result of an overflow or restricts it to its extremum value.

### Set Option

**User interface** (desktop products only): In the **Configuration** pane, the option is on the **Check Behavior** node.

**Command line and options file**: Use the option `-signed-integer-overflows`. See “Command-Line Information” (Polyspace Code Prover).

### Why Use This Option

Use this option to specify whether to check for signed integer overflows and to specify the assumptions the analysis makes following an overflow.

## Settings

**Default:** `forbid`

`forbid`

Polyspace flags signed integer overflows. If the **Overflow** check on an operation is:

- Red, Polyspace does not analyze the remaining code in the current scope.
- Orange, Polyspace analyzes the remaining code in the current scope. Polyspace considers that:
  - After a positive **Overflow**, the result of the operation has an upper bound. This upper bound is the maximum value allowed by the type of the result.
  - After a negative **Overflow**, the result of the operation has a lower bound. This lower bound is the minimum value allowed by the type of the result.

This behavior conforms to the ANSI C (ISO C++) standard.

In the following code, `j` has values in the range `[1..`$2^{31}$`-1]` before the orange overflow. Polyspace considers that `j` has even values in the range `[2 .. 2147483646]` after the overflow. Polyspace does not analyze the `printf()` statement after the red overflow.

```
#include<stdio.h>

int getVal();

void func1()
{
    int i = 1;
    i = i << 30;
    // Result of * operation overflows
    i = i * 2;
    // Remaing code in current scope not analyzed
    printf("%d", i);
}
void func2()
{

    int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^31-1]
        // Result of * operation may overflow
        j = j * 2;
        // Range of j: even values in [2 .. 2147483646]
        printf("%d", j);
    }
}
```

Note that tooltips on operations with signed integers show `(result is truncated)` to indicate the analysis mode. The message appears even if the **Overflow** check is green.

`allow`

Polyspace does not flag signed integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow.

In this code, the analysis does not flag any overflow in the code. However, the range of `j` wraps around to even values in the range $[-2^{31}..2]$ or $[2..2^{31}-2]$ and the value of `i` wraps around to $-2^{31}$.

```
#include<stdio.h>

int getVal();

void func1()
{
    int i = 1;
    i = i << 30;
    // i = 2^30
    i = i * 2;
    // i = -2^31
    printf("%d", i);
}
void func2()
{

    int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^31-1]
        j = j * 2;
        // Range of j: even values in [-2^31..2] or [2..2^31-2]
        printf("%d", j);
    }
}
```

Note that tooltips on operations with signed integers show `(result is wrapped)` to indicate the analysis mode. The message appears even if the analysis in this mode does not flag signed integer overflows.

`warn-with-wrap-around`

Polyspace flags signed integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow.

In the following code, `j` has values in the range `[1..`$2^{31}$`-1]` before the orange overflow. Polyspace considers that `j` has even values in the range `[-`$2^{31}$`..2] or [2..`$2^{31}$`-2]` after the overflow.

Similarly, `i` has value $2^{30}$ before the red overflow and value $-2^{31}$ after it .

```
#include<stdio.h>

int getVal();

void func1()
{
    int i = 1;
    i = i << 30;
    // i = 2^30
    // Result of * operation overflows
    i = i * 2;
    // i = -2^31
    printf("%d", i);
}
void func2()
{

    int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^31-1]
        // Result of * operation may overflow
        j = j * 2;
        // Range of j: even values in [-2^31..2] or [2..2^31-2]
        printf("%d", j);
    }
}
```

Note that tooltips on operations with signed integers show `(result is wrapped)` to indicate the analysis mode. The message appears even if the **Overflow** check is green.

In wrap-around mode, an overflowing value propagates and can lead to a similar overflow several lines later. By default, Code Prover shows only the first of similar overflows. To see all overflows, use the option `-show-similar-overflows`.

## Tips

- To check for overflows on conversions from unsigned to signed integers of the same size, set **Overflow mode for unsigned integer** to `forbid` or `warn-with-wrap-around`. If you allow unsigned integer overflows, Polyspace does not flag overflows on conversions and wraps the result of an overflow, even if you check for signed integer overflows.
- In Polyspace Code Prover, overflowing signed constants are wrapped around. This behavior cannot be changed by using the options. If you want to detect overflows with signed constants, use the Polyspace Bug Finder checker `Integer constant overflow`.

## Command-Line Information

**Parameter:** `-signed-integer-overflows`
**Value:** `forbid | allow | warn-with-wrap-around`
**Default:** `forbid`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-signed-integer-overflows allow`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-signed-integer-overflows allow`

## Version History

**Introduced in R2018b**

## See Also

`Overflow mode for unsigned integer (-unsigned-integer-overflows)` | `-show-similar-overflows` | `Overflow`

### Topics

“Specify Polyspace Analysis Options”
“Modify or Disable Code Prover Run-Time Checks” (Polyspace Code Prover)

# Overflow mode for unsigned integer (`-unsigned-integer-overflows`)

Specify whether result of overflow is wrapped around or truncated

## Description

*This option affects a Code Prover analysis only.*

Specify whether Polyspace flags unsigned integer overflows and whether the analysis wraps the result of an overflow or restricts it to its extremum value.

### Set Option

**User interface** (desktop products only): In the **Configuration** pane, the option is on the **Check Behavior** node under **Code Prover Verification**.

**Command line and options file**: Use the option `-unsigned-integer-overflows`. See “Command-Line Information” (Polyspace Code Prover).

### Why Use This Option

Use this option to specify how Polyspace Code Prover reacts to unsigned integer overflows. You can choose to flag an unsigned integer overflow and treat the code following this issue as compromised. Alternatively, you might choose to allow unsigned integer overflows. Depending on the value of this option, Polyspace makes different assumptions when analyzing the code that comes after an unsigned overflow.

## Settings

**Default:** `allow`

`forbid`

Polyspace flags unsigned integer overflows. If the **Overflow** check on an operation is:

- Red, Polyspace does not analyze the remaining code in the current scope.
- Orange, Polyspace analyzes the remaining code in the current scope. Polyspace considers that:
  - After a positive **Overflow**, the result of the operation has an upper bound. This upper bound is the maximum value allowed by the type of the result.
  - After a negative **Overflow**, the result of the operation has a lower bound. This lower bound is the minimum value allowed by the type of the result.

In the following code, `j` has values in the range `[1..`$2^{32}$`-1]` before the orange overflow. Polyspace considers that `j` has even values in the range `[2 .. 4294967294]` after the overflow. Polyspace does not analyze the `printf()` statement after the red overflow.

```
#include<stdio.h>

unsigned int getVal();

void func1()
{
    unsigned int i = 1;
    i = i << 31;
    // Result of * operation overflows
    i = i * 2;
    // Remaing code in current scope not analyzed
    printf("%u", i);
}
void func2()
{

    unsigned int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^32-1]
        // Result of * operation may overflow
        j = j * 2;
        // Range of j: even values in [2 .. 4294967294]
        printf("%u", j);
    }
}
```

Note that tooltips on operations with unsigned integers show `(result is truncated)` to indicate the analysis mode. The message appears even if the **Overflow** check is green.

`allow`

Polyspace does not flag unsigned integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow. For instance, `MAX_INT + 1` wraps to `MIN_INT`. This behavior conforms to the ANSI C (ISO C++) standard.

In this code, the analysis does not flag any overflow in the code. However, the range of `j` wraps around to even values in the range $[0..2^{32}-2]]$ and the value of `i` wraps around to `0`.

```
#include<stdio.h>

unsigned int getVal();

void func1()
{
    unsigned int i = 1;
    i = i << 31;
    // i = 2^31
    i = i * 2;
    // i = 0
    printf("%u", i);
}
void func2()
{

    unsigned int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^32-1]
        j = j * 2;
        // Range of j: even values in [0 .. 4294967294]
        printf("%u", j);
    }
}
```

Note that tooltips on operations with unsigned integers show `(result is wrapped)` to indicate the analysis mode. The message appears even if the analysis does not flag unsigned integer overflows.

`warn-with-wrap-around`

Polyspace flags unsigned integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow. For instance, `INT_MAX + 1` wraps to `0`.

In the following code, `j` has values in the range [1..$2^{32}$-1] before the orange overflow. Polyspace considers that `j` has even values in the range `[0 .. 4294967294]` after the overflow.

Similarly, `i` has value $2^{31}$ before the red overflow and value `0` after it.

```
#include<stdio.h>

unsigned int getVal();

void func1()
{
    unsigned int i = 1;
    i = i << 31;
    // i = 2^31
    i = i * 2;
    // i = 0
    printf("%u", i);
}
void func2()
{

    unsigned int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^32-1]
        j = j * 2;
        // Range of j: even values in [0 .. 4294967294]
        printf("%u", j);
    }
}
```

Note that tooltips on operations with unsigned integers show `(result is wrapped)` to indicate the analysis mode. The message appears even if the **Overflow** check is green.

In wrap-around mode, an overflowing value propagates and can lead to a similar overflow several lines later. By default, Code Prover shows only the first of similar overflows. To see all overflows, use the option `-show-similar-overflows`.

## Tips

- To check for overflows on conversions from unsigned to signed integers of the same size, set **Overflow mode for unsigned integer** to `forbid` or `warn-with-wrap-around`. If you allow unsigned integer overflows, Polyspace does not flag overflows on conversions and wraps the result of an overflow, even if you check for signed integer overflows.
- In Polyspace Code Prover, overflowing unsigned constants are wrapped around. This behavior cannot be changed by using the options. If you want to detect overflows with unsigned constants, use the Polyspace Bug Finder checker `Unsigned integer constant overflow`.
- Code Prover does not show an overflow on bitwise operations on unsigned variables or `sbit`-s, for instance, in this example:

  ```
  volatile unsigned char Y;
  Y = ~Y;
  ```

  The verification considers that such bitwise operations are deliberate on your part and you intend an automatic wrap-around in case the result of the operation overflows.

## Command-Line Information

**Parameter:** `-unsigned-integer-overflows`
**Value:** `forbid` | `allow` | `warn-with-wrap-around`

**Default:** `allow`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-unsigned-integer-overflows allow`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-unsigned-integer-overflows allow`

## Version History

**Introduced in R2018b**

## See Also

`Overflow mode for signed integer (-signed-integer-overflows)` | `-show-similar-overflows` | `Overflow`

### Topics

"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks" (Polyspace Code Prover)

# Allow incomplete or partial allocation of structures (-size-in-bytes)

Allow a pointer with insufficient memory buffer to point to a structure

## Description

*This option affects a Code Prover analysis only.*

Specify that the verification must allow dereferencing a pointer that points to a structure but has a sufficient buffer for only some of the structure's fields.

This type of pointer results when a pointer to a smaller structure is cast to a pointer to a larger structure. The pointer resulting from the cast has sufficient buffer for only some fields of the larger structure.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

**Command line and options file**: Use the option `-size-in-bytes`. See "Command-Line Information" on page 2-300.

### Why Use This Option

Use this option to relax the check for illegally dereferenced pointers. You can point to a structure even when the buffer allowed for the pointer is not sufficient for all the structure fields.

## Settings

☑ On

When a pointer with insufficient buffer is dereferenced, Polyspace does not produce an **Illegally dereferenced pointer** error, as long as the dereference occurs within allowed buffer.

For instance, in the following code, the pointer `p` has sufficient buffer for the first two fields of the structure `BIG`. Therefore, with the option on, Polyspace considers that the first two dereferences are valid. The third dereference takes `p` outside its allowed buffer. Therefore, Polyspace produces an **Illegally dereferenced pointer** error on the third dereference.

```
#include <stdlib.h>

typedef struct _little { int a; int b; } LITTLE;
typedef struct _big { int a; int b; int c; } BIG;

void main(void) {
   BIG *p = malloc(sizeof(LITTLE));

   if (p!= ((void *) 0) ) {
      p->a = 0 ;
```

```
          p->b = 0 ;
          p->c = 0 ;   // Red IDP check
         }
    }
```

□ Off (default)

Polyspace does not allow dereferencing a pointer to a structure if the pointer does not have sufficient buffer for all fields of the structure. It produces an **Illegally dereferenced pointer** error the first time you dereference the pointer.

For instance, in the following code, even though the pointer `p` has sufficient buffer for the first two fields of the structure `BIG`, Polyspace considers that dereferencing `p` is invalid.

```
#include <stdlib.h>

typedef struct _little { int a; int b; } LITTLE;
typedef struct _big { int a; int b; int c; } BIG;

void main(void) {
   BIG *p = malloc(sizeof(LITTLE));

   if (p!= ((void *) 0) ) {
      p->a = 0 ;   // Red IDP check
      p->b = 0 ;
      p->c = 0 ;
   }
}
```

## Tips

- If you do not turn on this option, you cannot point to the field of a partially allocated structure.

  For instance, in the preceding example, if you do not turn on the option and perform the assignment

  ```
  int *ptr = &(p->a);
  ```

  Polyspace considers that the assignment is invalid. If you dereference `ptr`, it produces an **Illegally dereferenced pointer** error.
- Using this option can slightly increase the number of orange checks.

## Command-Line Information

**Parameter:** `-size-in-bytes`
**Default**: Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-size-in-bytes`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-size-in-bytes`

## See Also

`Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)` | `Illegally dereferenced pointer`

### Topics

"Specify Polyspace Analysis Options"

“Modify or Disable Code Prover Run-Time Checks” (Polyspace Code Prover)

# Subnormal detection mode (`-check-subnormal`)

Detect operations that result in subnormal floating-point values

## Description

*This option affects a Code Prover analysis only.*

Specify that the verification must check floating-point operations for subnormal results.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

**Command line and options file**: Use the option `-check-subnormal`. See “Command-Line Information” on page 2-304.

### Why Use This Option

Use this option to detect floating-point operations that result in subnormal values.

Subnormal numbers have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the significand. The presence of subnormal numbers indicates loss of significant digits. This loss can accumulate over subsequent operations and eventually result in unexpected values. Subnormal numbers can also slow down the execution on targets without hardware support.

## Settings

**Default:** `allow`

`allow`

The verification does not check operations for subnormal results.

`forbid`

The verification checks for subnormal results.

The verification stops the execution path with the subnormal result and prevents subnormal values from propagating further. Therefore, in practice, you see only the first occurrence of the subnormal value.

`warn-all`

The verification checks for subnormal results and highlights all occurrences of subnormal values. Even if a subnormal result comes from previous subnormal values, the result is highlighted.

The verification continues even if the check is red.

`warn-first`

The verification checks for subnormal results but only highlights first occurrences of subnormal values. If a subnormal value propagates to further subnormal results, those subsequent results are not highlighted.

The verification continues even if the check is red.

For details of the result colors in each mode, see `Subnormal float`.

## Tips

- If you want to see only those operations where a subnormal value originates from non-subnormal operands, use the `warn-first` mode.

  For instance, in the following code, `arg1` and `arg2` are unknown. The verification assumes that they can take all values allowed for the type `double`. This assumption can lead to subnormal results from certain operations. If you use the `warn-first` mode, the first operation causing the subnormal result is highlighted.

| **warn-all** | **warn-first** |
|---|---|
| `void func (double arg1, double arg2)`<br>`{`<br>`      double difference1 = arg1 - arg2;`<br>`      double difference2 = arg1 - arg2;`<br>`      double val1 = difference1 * 2;`<br>`      double val2 = difference2 * 2;`<br>`}`<br>In this example, all four operations can have subnormal results. The four checks for subnormal results are orange. | `void func (double arg1, double arg2)`<br>`{`<br>`      double difference1 = arg1 - arg2;`<br>`      double difference2 = arg1 - arg2;`<br>`      double val1 = difference1 * 2;`<br>`      double val2 = difference2 * 2;`<br>`}`<br>In this example, `difference1` and `difference2` can be subnormal if `arg1` and `arg2` are sufficiently close. The first two checks for subnormal results are orange. `val1` and `val2` cannot be subnormal unless `difference1` and `difference2` are subnormal. The last two checks for subnormal results are green.<br>Through red/orange checks, you see only the first instance where a subnormal value appears. You do not see red/orange checks from those subnormal values propagating to subsequent operations. |

- If you want to see where a subnormal value originates and do not want to see subnormal results arising from the same cause more than once, use the `forbid` mode.

  For instance, in the following code, `arg1` and `arg2` are unknown. The verification assumes that they can take all values allowed for the type `double`. This assumption can lead to subnormal results for `arg1-arg2`. If you use the `forbid` mode and perform the operation `arg1-arg2` twice in succession, only the first operation is highlighted. The second operation is not highlighted because the subnormal result for the second operation arises from the same cause as the first operation.

| **warn-all** | **forbid** |
|---|---|
| `void func (double arg1, double arg2)`<br>`{`<br>`      double difference1 = arg1 - arg2;`<br>`      double difference2 = arg1 - arg2;`<br>`      double val1 = difference1 * 2;`<br>`      double val2 = difference2 * 2;`<br>`}`<br>In this example, all four operations can have subnormal results. The four checks for subnormal results are orange. | `void func (double arg1, double arg2)`<br>`{`<br>`      double difference1 = arg1 - arg2;`<br>`      double difference2 = arg1 - arg2;`<br>`      double val1 = difference1 * 2;`<br>`      double val2 = difference2 * 2;`<br>`}`<br>In this example, `difference1` can be subnormal if `arg1` and `arg2` are sufficiently close. The first check for subnormal results is orange. Following this check, the verification excludes from consideration:<br>• The close values of `arg1` and `arg2` that led to the subnormal value of `difference1`.<br>In the subsequent operation `arg1 - arg2`, the check is green and `difference2` is not subnormal. The result of the check on `difference2 * 2` is green for the same reason.<br>• The subnormal value of `difference1`.<br>In the subsequent operation `difference1 * 2`, the check is green. |

## Command-Line Information

**Parameter:** `-check-subnormal`
**Value:** `allow` | `warn-first` | `warn-all` | `forbid`
**Default:** `allow`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-check-subnormal forbid`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-check-subnormal forbid`

# Version History

**Introduced in R2016b**

## See Also

### Polyspace Results

`Subnormal float`

### Topics

"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks" (Polyspace Code Prover)

# Detect uncalled functions (-uncalled-function-checks)

Detect functions that are not called directly or indirectly from main or another entry point function

## Description

*This option affects a Code Prover analysis only.*

Detect functions that are not called directly or indirectly from main or another entry point function during run-time.

A function can be uncalled because of several reasons including the following:

- The function is actually not called.
- The call site occurs in dead code.
- The call site follows a red check. See “Code Prover Analysis Following Red and Orange Checks” (Polyspace Code Prover).
- The call occurs via a function pointer and Polyspace is unable to determine which function it points to.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** node.

**Command line and options file**: Use the option -uncalled-function-checks. See “Command-Line Information” on page 2-306.

### Why Use This Option

Typically, after verification, the **Dashboard** pane in the Polyspace user interface shows functions that are not called during verification and therefore not analyzed. However, you do not see them in your analysis results or reports. You cannot comment on them or justify them.

If you want to see these uncalled functions in your analysis results and reports, use this option.

## Settings

**Default:** none

none

The Code Prover analysis excludes checks for uncalled functions.

never-called

The Code Prover analysis checks for functions that are defined but not called.

called-from-unreachable

The Code Prover analysis checks for functions that are defined and called from an unreachable part of the code.

`all`

The Code Prover analysis checks for functions that are:

- Defined but not called
- Defined and called from an unreachable part of the code.

## Command-Line Information

**Parameter:** `-uncalled-function-checks`
**Value:** `none` | `never-called` | `called-from-unreachable` | `all`
**Default**: `none`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-uncalled-function-checks all`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-uncalled-function-checks all`

## See Also

`Function not called` | `Function not reachable`

### Topics

“Specify Polyspace Analysis Options”
“Modify or Disable Code Prover Run-Time Checks” (Polyspace Code Prover)

# Calculate stack usage (`-stack-usage`)

Compute and display the estimates of stack usage

## Description

*This option applies to Code Prover only.*

Specify that Polyspace must estimate and display the stack usage of your source code. The estimates include:

- `Maximum Stack Usage`
- `Minimum Stack Usage`
- `Program Maximum Stack Usage`
- `Program Minimum Stack Usage`
- `Higher Estimate of Size of Local Variables`
- `Lower Estimate of Size of Local Variables`

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Check Behavior** subnode of the **Code Prover Verification** node.

**Command line and options file**: Use the option `-stack-usage`. See “Command-Line Information” on page 2-196.

### Why Use This Option

Estimating the stack usage of your code is necessary because:

- Underestimating the stack usage of your code results in a stack overflow.
- Overestimating the stack usage results in wasted memory.

Obtain the estimated stack usage of your code by using the option `-stack-usage` to avoid a stack overflow or wasted memory. This option is especially important for safety critical applications where the available stack size must accommodate the worst-case stack usage.

## Settings

☑ On

Polyspace computes and displays stack usage metrics on the **Results List** pane.

□ Off (default)

Polyspace does not compute stack usage metrics.

## Tips

- If you want to compute only the stack usage of your code, run verification up to the `Source Compliance Checking` phase. See `Verification level (-to)`.
- A Code Prover analysis computes the stack usage metrics after the source compliance checking phase. If you stop a Code Prover verification before source compliance checking, the stack usage metrics are not reported.
- This option calculates code metrics that are related to stack usage. Use Bug Finder to calculate the other code metrics. See `Calculate code metrics (-code-metrics)`.
- Using this option together with `-code-metrics` option results in an error. The option `-code-metrics` will be removed from Code Prover in a future release. To compute the code metrics, use Bug Finder instead. See "Migrate Code Prover Workflows for Checking Coding Standards and Code Metrics to Bug Finder".
- If a function is not defined, Code Prover considers zero contribution to the stack size when this function is called. You can explicitly specify a stack size contribution from such stubbed functions using the option `-code-behavior-specifications`.

## Command-Line Information

**Parameter:** `-stack-usage`
**Default:** Off
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-stack-usage`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-stack-usage`

# Version History

**Introduced in R2022a**

## See Also

`-code-behavior-specifications` | `Higher Estimate of Size of Local Variables` | `Lower Estimate of Size of Local Variables` | `Maximum Stack Usage` | `Minimum Stack Usage` | `Program Maximum Stack Usage` | `Program Minimum Stack Usage`

### Topics

"Compute Code Complexity Metrics Using Polyspace"
"Code Metrics"
"Migrate Code Prover Workflows for Checking Coding Standards and Code Metrics to Bug Finder"
"Determination of Program Stack Usage" (Polyspace Code Prover)

# Sensitivity context (-context-sensitivity)

Store call context information to identify function call that caused errors

## Description

*This option affects a Code Prover analysis only.*

Specify the functions for which the verification must store call context information. If the function is called multiple times, using this option helps you to distinguish between the different calls.

### Set Option

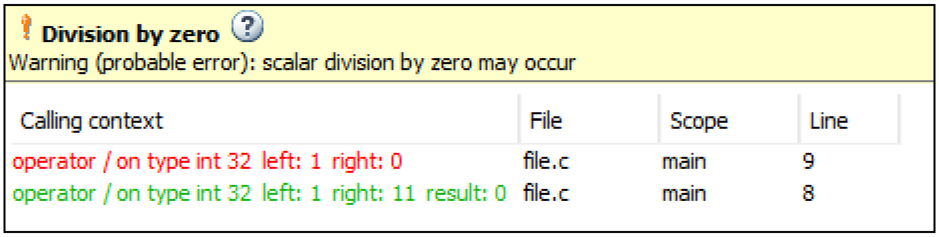
**User interface** (desktop products only): In your project configuration, the option is available on the **Precision** node.

**Command line and options file**: Use the option `-context-sensitivity`. See “Command-Line Information” (Polyspace Code Prover).

### Why Use This Option

Suppose a function is called twice in your code. The check color on each operation in the function body is a combined result of both calls. If you want to distinguish between the colors in the two calls, use this option.

For instance, if a function contains a red or orange check and a green check on the same operation for two different calls, the software combines the contexts and displays an orange check on the operation. If you use this option, the check turns dark orange and the result details show the color of the check for each call.



**Division by zero**

Warning (probable error): scalar division by zero may occur

| Calling context | File | Scope | Line |
|---|---|---|---|
| operator / on type int 32 left: 1 right: 0 | file.c | main | 9 |
| operator / on type int 32 left: 1 right: 11 result: 0 | file.c | main | 8 |

## Settings

**Default:** `none`

`none`

The software does not store call context information for functions.

`auto`

The software stores call context information for checks in:

- Functions that form the leaves of the call tree. These functions are called by other functions, but do not call functions themselves.

- Small functions. The software uses an internal threshold to determine whether a function is small.

`custom`

The software stores call context information for functions that you specify. To enter the name of a function, click ✚.

## Tips

- If you select this option, you do not see tooltips in the body of the functions that benefit from this option (and keep the call contexts separate).
- If you select this option, the analysis can show some code operations in grey (unreachable code) even when you can identify execution paths leading to the operations. In this case, the grey code indicates operations that might be unreachable only in a particular call context.

  For instance, suppose this function is called with the arguments -1 and 1 :

  ```
  int isPositive (int num) {
     if(num < 0)
        return 0;
     return 1;
  }
  ```

  If you use the option with this function as argument, there are two unreachable code checks:

  - The check on `if` is grey because when the function is called with argument -1, the `if` condition is always true.
  - The check on the code inside the `if` branch is grey because when the function is called with argument 1, the `if` condition is always false.

  Each unreachable code check indicates code that is unreachable only in a particular call context. You see the call context in the result details.

## Command-Line Information

**Parameter:** `-context-sensitivity`
**Value:** `*function1*[,*function2*,...]`
**Default:** `none`
**Example (Code Prover):** `polyspace-code-prover -sources *file_name* -context-sensitivity myFunc1,myFunc2`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources *file_name* -context-sensitivity myFunc1,myFunc2`

To allow the software to determine which functions receive call context storage, use the option `-context-sensitivity-auto`.

## See Also

### Topics

“Specify Polyspace Analysis Options”
“Identify Function Call with Run-Time Error” (Polyspace Code Prover)

# Improve precision of interprocedural analysis (-path-sensitivity-delta)

Avoid certain verification approximations for code with fewer lines

## Description

*This option affects a Code Prover analysis only.*

For smaller code, use this option to improve the precision of cross-functional analysis.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Precision** node.

**Command line and options file**: Use the option `-path-sensitivity-delta`. See “Command-Line Information” on page 2-311.

### Why Use This Option

Use this option to avoid certain software approximations on execution paths. Avoiding these approximations results in fewer orange checks but a much longer verification time.

For instance, for deep function call hierarchies or nested conditional statements, to complete verification in a reasonable amount of time, the software combines many execution paths and stores less information at each stage of verification. If you use this option, the software stores more information about the execution paths, resulting in a more precise verification.

## Settings

**Default:** Off

Enter a positive integer to turn on this option.

Entering a higher value leads to a greater number of proven results, but also increases verification time exponentially. For instance, a value of 10 can result in very long verification times.

## Tips

Use this option only when you have less than 1000 lines of code.

## Command-Line Information

**Parameter:** `-path-sensitivity-delta`
**Value:** Positive integer
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-path-sensitivity-delta 1`

**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-path-sensitivity-delta 1`

## See Also

### Topics

"Specify Polyspace Analysis Options"
"Improve Verification Precision" (Polyspace Code Prover)

# Precision level (`-O0 | -O1 | -O2 | -O3`)

Specify a precision level for the verification

## Description

*This option affects a Code Prover analysis only.*

Specify the precision level that the verification must use.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Precision** node.

**Command line and options file**: Use the option `-O#`, for instance, `-O0` or `-O1`. See “Command-Line Information” on page 2-314.

### Why Use This Option

Higher precision leads to greater number of proven results but also requires more verification time. Each precision level corresponds to a different algorithm used for verification.

In most cases, you see the optimal balance between precision and verification time at level 2.

## Settings

**Default:** 2

`0`

This option corresponds to a static interval verification.

`1`

This option corresponds to a more complex static interval verification.

`2`

This option corresponds to a complex polyhedron model of domain values with additional precision for interprocedural analysis depending on the option `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

`3`

This option is only suitable for code having less than 1000 lines. Using this option, the percentage of proven results can be very high.

## Tips

- For best results in reasonable time, use the default level 2. If the verification takes a long time, reduce precision. However, the number of unproven checks can increase. Likewise, to reduce orange checks, you can improve your precision. But the verification can take significantly longer time.

- The precision levels 2 and below begin to take effect only from verification levels higher than `Software Safety Analysis level 0`. See also `Verification level (-to)`.

  For instance, to reduce analysis time, you might have reduced the verification level to `Software Safety Analysis level 0`. Do not try to reduce the precision level below 2 to lower the analysis time further.

  Note that algorithms used in precision level 3 can also apply to the verification level `Software Safety Analysis level 0`.

## Command-Line Information

**Parameter:** `-O0` | `-O1` | `-O2` | `-O3`
**Default:** `-O2`
**Example (Code Prover):** `polyspace-code-prover -sources` *file_name* `-O1`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *file_name* `-O1`

## See Also

`Verification level (-to)` | `Specific precision (-modules-precision)`

### Topics

"Specify Polyspace Analysis Options"
"Improve Verification Precision" (Polyspace Code Prover)

# Specific precision (`-modules-precision`)

Specify source files you want to verify at higher precision than the remaining verification

## Description

*This option affects a Code Prover analysis only.*

Specify source files that you want to verify at a precision level higher than that for the entire verification.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Precision** node. See “Dependency” on page 2-315 for other options you must also enable.

**Command line and options file**: Use the option `-modules-precision`. See “Command-Line Information” on page 2-315.

### Why Use This Option

If a specific file is verified imprecisely leading to many orange checks in the file and elsewhere, you can improve the precision for that file.

Note that increasing precision also increases verification time.

## Settings

**Default:** All files are verified with the precision you specified using **Precision** > **Precision level**.

Click ✚ to enter the name of a file without the extension `.c` and the corresponding precision level.

## Dependency

This option is available only if you set `Source code language (-lang)` to `C` or `C-CPP`.

## Command-Line Information

**Parameter:** `-modules-precision`
**Value:** *`file`*`:O0` | *`file`*`:O1` | *`file`*`:O2` | *`file`*`:O3`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-O1 -modules-precision My_File:O2`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-O1 -modules-precision My_File:O2`

## See Also

`Precision level (-O)`

**Topics**

"Specify Polyspace Analysis Options"
"Improve Verification Precision" (Polyspace Code Prover)

# Verification level (-to)

Specify number of times the verification process runs on your code

## Description

*This option affects a Code Prover analysis only.*

Specify the number of times the Polyspace verification process runs on your source code. Each run can lead to greater number of proven results but also requires more verification time.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Precision** node.

**Command line and options file**: Use the option `-to`. See “Command-Line Information” on page 2-318.

### Why Use This Option

There are many reasons you might want to increase or decrease the verification level. For instance, If you see many orange checks after verification, try increasing the verification level. If your analysis takes longer than expected, try decreasing the verification level.

In most cases, the optimal balance between precision and verification time is at level 2.

## Settings

**Default:** `Software Safety Analysis level 2`

`Source Compliance Checking`

Polyspace checks for compilation errors only. Most coding rule violations are found at this level.

`Software Safety Analysis level 0`

The verification process performs some simple analysis. The analysis is designed to reach completion despite complexities in the code.

If the verification gets stuck at a higher level, try running to this level and review the results.

`Software Safety Analysis level 1`

The verification process analyzes each function once with algorithms whose complexity depends on the precision level. See `Precision level (-O)`. The analysis starts from the top of the function call hierarchy (an actual or generated `main` function) and propagates to the leaves of the call hierarchy.

`Software Safety Analysis level 2`

The verification process analyzes each function twice. In the first pass, the analysis propagates from the top of the function call hierarchy to the leaves. In the second pass, the analysis propagates from the leaves back to the top. Each pass uses information gathered from the previous pass.

Use this option for most accurate results in reasonable time.

`Software Safety Analysis level 3`

The verification process runs three times on each function: from the top of the function call hierarchy to the leaves, from the leaves to the top, and from the top to the leaves again. Each pass uses information gathered from the previous pass.

`Software Safety Analysis level 4`

The verification process runs four passes on each function: from the top of the function call hierarchy to the leaves twice. Each pass uses information gathered from the previous pass.

`other`

If you use this option, Polyspace verification will make 20 passes unless you stop it manually.

## Tips

- If the verification takes too long, use a lower **Verification level**. For best results, use the option `Software Safety Analysis level 2`.
- Fix red errors and gray code before rerunning the verification with higher verification levels.
- In some cases, if the results have reached the maximum precision at an earlier level, the verification stops and does not proceed to the level that you specify. If a higher verification level fails because the verification runs out of memory, but results are available at a lower level, Polyspace displays the results from the lower level.
- Use the option `Other` sparingly since it might increase verification time by an unreasonable amount.
- When running Polyspace analyses on a remote server, the source phase takes place on your local computer. If the **Verification Level** is set to `Source Compliance Checking`, do not use `-batch` to submit jobs to a remote server from a desktop. See `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.
- If you want to see only global variable sharing and usage, use `Show global variable sharing and usage only (-shared-variables-mode)` to run a less extensive analysis.

## Command-Line Information

**Parameter:** `-to`
**Value:** `compile` | `pass0` | `pass1` | `pass2` | `pass3` | `pass4` | `other`
**Default:** `pass2`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-to pass2`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-to pass2`

You can also use these additional values not available in the user interface:

- C projects: `c-to-il` (C to intermediate language conversion phase)
- C++ projects: `cpp-to-il` (C++ to intermediate language conversion phase), `cpp-normalize` (C ++ compilation), `cpp-link` (C++ compilation)

Use these values only if you have specific reasons to do so. For instance, to generate a blank constraints (DRS) template for C++ projects, run an analysis up to the compilation by using `cpp-link` or `cpp-normalize`.

The values `cpp-link` and `cpp-normalize` will be removed in a future release. Use `compile` instead.

## Examples

### Reduce Number of Orange Checks by Increasing Verification Level

Running a Code Prover verification at a higher verification level reduces the number of orange checkers.

Run a Code Prover verification on this code while setting the precision to `Software Safety Analysis level 0`.

```
extern int tab[];

int main() {

    int i = tab[3];
    int j = tab[1];

    if (i > j) {
        int l = i-j;
        assert(l > 0);
     }
}
```

There is an orange check in the `assert` statement.

To resolve the orange check, run the analysis at a higher precision. For instance, run another verification on the preceding code while setting the precision to `Software Safety Analysis level 1`.

```
extern int tab[];

int main() {

    int i = tab[3];
    int j = tab[1];

    if (i > j) {
        int l = i-j;
        assert(l > 0);
     }
}
```

The `assert` statement now has a green check.

## See Also

`Precision level (-O)` | `Show global variable sharing and usage only (-shared-variables-mode)`

**Topics**

"Specify Polyspace Analysis Options"
"Improve Verification Precision" (Polyspace Code Prover)

# Verification time limit (`-timeout`)

Specify a time limit on your verification

## Description

Specify a time limit for the verification in hours. If the verification does not complete within that limit, it stops.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Precision** node.

**Command line and options file**: Use the option `-timeout`. See “Command-Line Information” on page 2-320.

### Why Use This Option

Use this option to impose a time limit on the verification.

By default, if an internal step in the verification lasts for more than 24 hours, the verification stops. You can use this option to reduce the time limit even further. Note that you can have verification results despite the verification timing out. For instance, if a step in Software Safety Analysis level 1 times out, you still get the results from level 0. See `Verification level (-to)`.

The option is useful only in very specific cases. Suppose your code has certain constructs that might slow down the verification. To check this, you can impose a time limit on the verification so that the verification stops if it takes too long.

Typically, Technical Support asks you to use this option as needed.

## Settings

Enter the time in hours. For fractions of an hour, specify decimal form.

## Command-Line Information

**Parameter:** `-timeout`
**Value:** *`time`*
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-timeout 5.75`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-timeout 5.75`
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-timeout 5.75`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-timeout 5.75`

## See Also

### Topics

“Specify Polyspace Analysis Options”

"Improve Verification Precision" (Polyspace Code Prover)

# Inline (-inline)

Specify functions that must be cloned internally for each function call

## Description

*This option affects a Polyspace Code Prover analysis only.*

Specify the functions that the verification must clone internally for each function call.

### Set Option

**User interface**, in desktop products only: In your project configuration, the `-inline` option is available on the **Scaling** node.

**Command line and options file**: Use the `-inline` option. See "Command-Line Information" on page 2-324.

### Why Use This Option

You use the `-inline` to internally clone a function for each function call. Doing so might reduce the complexity of the analysis in Polyspace Code Prover. Consider this code:

```
void foo(int* p);
void bar(){
    int* intP, intP2;
    foo(intP);
    foo(intP2)
}
```

In this code, Polyspace analyzes the functions `foo()` and `bar()`. When the option `-inline` Is not specified, the analysis keeps track of three objects: the pointers `intP`, `intP2`, and the parameter `p`. If you specify the option `-inline`, Polyspace internally clones `foo` for each function call. The code effectively becomes:

```
void foo_clone1(int* p);
void foo_clone2(int* p);
void bar(){
    int* intP, intP2;
    foo_clone1(intP);
    foo_clone2(intP2)
}
```

Analyzing the function `foo_clone1()` requires Polyspace to keep track of only two objects, `intP` and `p`, instead of three objects. The same is true for the function `foo_clone2()`. By using `-inline`, you reduce the complexity of the analysis and increase the amount of code to analyze.

Using `-inline` might reduce the complexity of your analysis while increasing the amount of code to analyze. This option might help you resolve scaling issues in specific situations. Technical Support might ask you to use this option for certain functions. Using this option changes the meaning of the colors of Polyspace Code Prover checks. See "Settings" (Polyspace Code Prover).

Avoid using this option to distinguish between the Code Prover check colors in the different calls to the same function. Instead, use the option `Sensitivity context (-context-sensitivity)`.

## Settings

**No Default**

Enter function names or choose from a list.

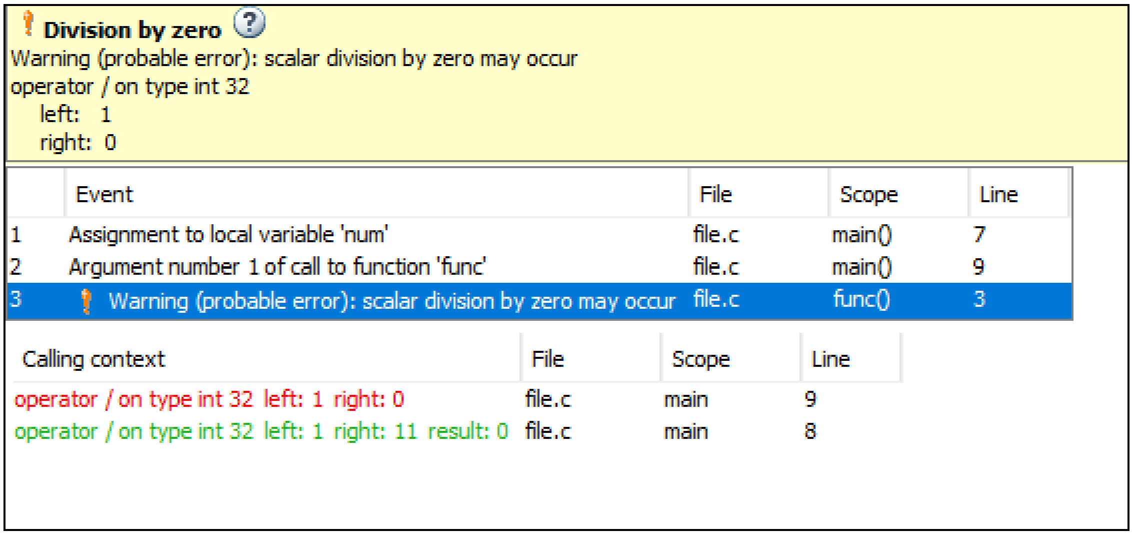
- Click ✚ to add a field and enter the function name.
- Click 🔍 to list functions in your code. Choose functions from the list.

The verification internally clones the function for each call. For instance, if you specify the function `func` for inlining and call `func` twice, the software internally creates two copies of `func` for verification.

However, for each run-time check in the function body, you see only one color in your verification results. The semantics of the check color differ from the normal specification.

| Check Type | Result color without the option `-inline` | Result color with the option `-inline` |
|---|---|---|
| Orange checks | If a function is called twice and an operation causes a definite error in one of the calls, the check color is orange. | If a function is called twice and an operation causes a definite error in one of the calls, the check color is dark orange. The check is shown with an orange exclamation mark in the results list. |
| Gray checks | If a function is called twice and an `if` statement branch is unreachable in one of the calls, the branch does not appear gray. | If a function is called twice and an `if` statement branch is unreachable in one of the calls, the branch appears gray. |

Below each check in an inlined function, you see information specific to each calling context. For instance, if an orange exclamation point appears next to **Division by zero** occurs because a specific function call leads to a definite division by zero, you can identify the call along with values resulting from that call.



**Division by zero**

Warning (probable error): scalar division by zero may occur
operator / on type int 32
left: 1
right: 0

| | Event | File | Scope | Line |
|---|---|---|---|---|
| 1 | Assignment to local variable 'num' | file.c | main() | 7 |
| 2 | Argument number 1 of call to function 'func' | file.c | main() | 9 |
| 3 | Warning (probable error): scalar division by zero may occur | file.c | func() | 3 |

| Calling context | File | Scope | Line |
|---|---|---|---|
| operator / on type int 32 left: 1 right: 0 | file.c | main | 9 |
| operator / on type int 32 left: 1 right: 11 result: 0 | file.c | main | 8 |

Do not use this option to understand results. Use this option only if a certain function causes scaling issues.

## Tips

- Choose functions to inline based on hints provided by the alias verification.
- Do not use this option for entry point functions, including `main`.
- Using this option can increase the number of gray **Unreachable code** checks.

  For example, in the following code, if you enter `max` for **Inline**, two **Unreachable code** checks appear for each call to `max`.

  ```
  int max(int a, int b) {
    return a > b ? a : b;
  }

  void main() {
    int i=3, j=1, k;
    k=max(i,j);
    i=0;
    k=max(i,j);
  }
  ```
- If you use the keyword `inline` before a function definition, place the definition in a header file and call the function from multiple source files, the result is the same as using the option **Inline**.
- For C++ code, this option applies to all overloaded methods of a class.

## Command-Line Information

**Parameter:** `-inline`
**Value:** `*function1*[,*function2*[,...]]`
**No Default**
**Example (Code Prover):** `polyspace-code-prover -sources *file_name* -inline func1,func2`

**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-inline func1,func2`

## See Also

### Topics

"Specify Polyspace Analysis Options"

# Depth of verification inside structures (`-k-limiting`)

Limit the depth of analysis for nested structures

## Description

*This option affects a Code Prover analysis only.*

Specify a limit to the depth of analysis for nested structures.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Scaling** node.

**Command line and options file**: Use the option `-k-limiting`. See “Command-Line Information” on page 2-326.

### Why Use This Option

Use this option if the analysis is slow because your code has a structure that is many levels deep.

Typically, you see a warning message when a structure with a deep hierarchy is slowing down the verification.

## Settings

**Default:** Full depth of nested structures is analyzed.

Enter a number to specify the depth of analysis for nested structures. For instance, if you specify 0, the analysis does not verify a structure inside a structure.

If you specify a number less than 2, the verification could be less precise.

## Command-Line Information

**Parameter:** `-k-limiting`
**Value:** *`positive integer`*
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-k-limiting 3`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-k-limiting 3`

## See Also

### Topics

“Specify Polyspace Analysis Options”

# Generate report

Specify whether to generate a report after the analysis

## Description

Specify whether to generate a report along with analysis results.

Depending on the format you specify, you can view this report using an external software. For example, if you specify the format `PDF`, you can view the report in a `pdf` reader.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Reporting** node.

**Command line and options file**: See “Command-Line Information” on page 2-328.

### Why Use This Option

You can generate a report from your analysis results for archiving purposes. You can provide this report to your management or clients as proof of code quality.

Using other analysis options, you can tailor the report content and format for your specific needs. See `Bug Finder and Code Prover report (-report-template)` and `Output format (-report-output-format)`.

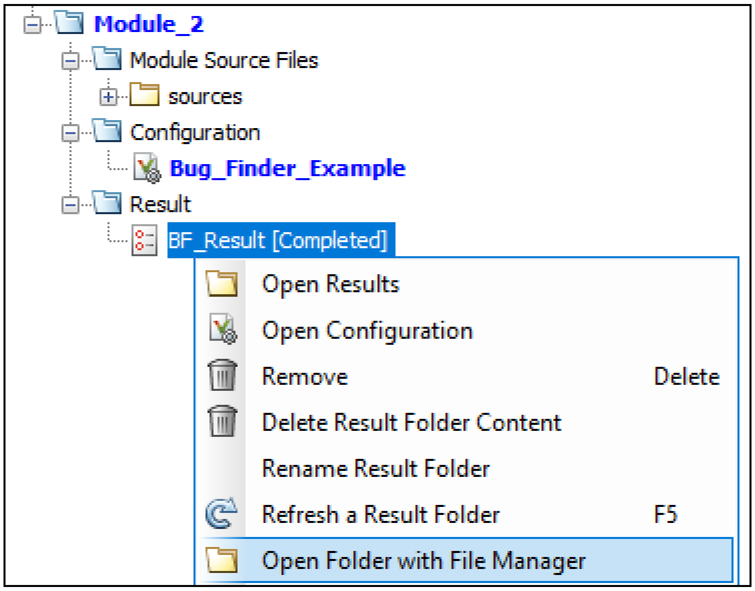
## Settings

☑ On

Polyspace generates an analysis report using the template and format you specify.

The report is stored in the `Polyspace-Doc` subfolder of your results folder.

In Polyspace desktop products, to open your results folder from the user interface, on the **Project Browser** pane, right-click the results node and select **Open Folder with File Manager**.



To change the results folder location, see “Contents of Polyspace Project and Results Folders”.

On the command-line, the results folder is the argument of the option `-results-dir`.

□ Off (default)

Polyspace does not generate an analysis report. You can still view your results in the Polyspace interface.

## Tips

This option allows you to specify report generation before starting an analysis.

To generate a report *after* an analysis is complete, in the user interface of the Polyspace desktop products, select **Reporting** > **Run Report**. Alternatively, at the command line, use the `polyspace-report-generator` command.

After analysis, you can also export the result as a text file for further customization. Use the option `-generate-results-list-file` with the `polyspace-report-generator` command.

## Command-Line Information

There is no command-line option to solely turn on the report generator. However, using the options `-report-template` for template and `-report-output-format` for output format automatically turns on the report generator.

## See Also

`Bug Finder and Code Prover report (-report-template)` | `Output format (-report-output-format)` | `polyspace-report-generator`

### Topics

“Specify Polyspace Analysis Options”
“Generate Reports from Polyspace Results”

# Bug Finder and Code Prover report (`-report-template`)

Specify template for generating analysis report

## Description

Specify template for generating analysis report.

`.rpt` files for the report templates are available in *`polyspaceroot`*`\toolbox\polyspace\psrptgen\templates\`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a`.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Reporting** node. You have separate options for Bug Finder and Code Prover analysis. See “Dependencies” on page 2-334 for other options you must also enable.

**Command line and options file**: Use the option `-report-template`. See “Command-Line Information” on page 2-335.

### Why Use This Option

Depending on the template that you use, the report contains information about certain types of results from the **Results List** pane. The template also determines what information is presented in the report and how the information is organized. See the template descriptions below.

## Settings – Bug Finder

**Default:** `BugFinderSummary`

`BugFinder`

The report lists:

- **Polyspace Bug Finder Summary**: Number of results in the project. The results are summarized by file. The files that are partially analyzed because of compilation errors are listed in a separate table.
- **Code Metrics**: Summary of the various code complexity metrics. For more information, see “Code Metrics”.
- **Coding Rules**: Coding rule violations in the source code. For each rule violation, the report lists the:
  - Rule number and description.
  - Function containing the rule violation.
  - Review information, such as **Severity**, **Status** and comments.
- **Defects**: Defects found in the source code. For each defect, the report lists the:

  - Function containing the defect.
  - Defect information on the **Result Details** pane.
  - Review information, such as **Severity**, **Status** and comments.
- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. If you configured your project for multitasking, this section also lists the **Concurrency Modeling Summary**. If your project has source files with compilation errors, these files are also listed.

  If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

`BugFinderSummary`

The report lists:

- **Polyspace Bug Finder Summary**: Number of results in the project. The results are summarized by file. The files that are partially analyzed because of compilation errors are listed in a separate table.
- **Code Metrics**: Summary of the various code complexity metrics. For more information, see “Code Metrics”.
- **Coding Rules Summary**: Coding rules along with number of violations.
- **Defect Summary**: Defects that Polyspace Bug Finder looks for. For each defect, the report lists the:
  - Defect group.
  - Defect name.
  - Number of instances of the defect found in the source code.
- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. If you configured your project for multitasking, this section also lists the **Concurrency Modeling Summary**. For more information, see “Complete List of Polyspace Bug Finder Analysis Engine Options”. If your project has source files with compilation errors, these files are also listed.

  If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

`CodeMetrics`

The report lists the following:

- **Code Metrics Summary**: Various quantities related to the source code. For more information, see “Code Metrics”.
- **Code Metrics Details**: Various quantities related to the source code with the information broken down by file and function.
- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. If you configured your project for multitasking, this section also lists the **Concurrency Modeling Summary**. If your project has source files with compilation errors, these files are also listed.

  If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

`CodingStandards`

The report contains separate chapters for each coding standard enabled in the analysis (for instance, MISRA C: 2012, CERT C, custom rules, and so on). Each chapter contains the following information:

- **Summary - Violations by File**: Graph showing each file with number of rule violations.
- **Summary - Violations by Rule**: Graph showing each rule with number of violations. If a rule is not enabled or not violated, it does not appear in the graph.
- **Summary for all Files**: Table showing each file with number of rule violations.
- **Summary for Enabled Guidelines** or **Summary for Enabled Rules**: Table showing each guideline or rule with number of violations.
- **Violations**: Tables listing each rule violation, along with information such as ID, function name, severity, status, and so on. One table is created per file.

An appendix lists the options used in the Polyspace analysis.

`SecurityCWE`

The report contains the same information as the `BugFinder` report. However, in the **Defects** chapter, an additional column lists the CWE™ rules mapped to each defect. The **Configuration Settings** appendix also includes a **Security Standard to Polyspace Result Map**.

`Metrics`

*Only available for results that you upload to the Polyspace Access interface.*

The report lists information useful to quality engineers and available on the Polyspace Access interface, including:

- Information about whether the project satisfies quality objectives
- Metrics about the whole project. For each metric, the report lists the quality threshold and whether the metric satisfies this threshold.
- Coding rule violations in the project. For each rule, the report lists the number of violations justified and whether the justifications satisfy quality objectives.
- Definite as well as possible run-time errors in the project. For each type of run-time error, the report lists the number of errors justified and whether the justifications satisfy quality objectives.

The appendices contain further details of Polyspace configuration settings, code metrics, coding rule violations, and run-time errors.

## Settings – Code Prover

**Default:** `Developer`

`CodeMetrics`

The report contains a summary of code metrics, followed by the complete metrics for an application.

`CodingStandards`

The report contains separate chapters for each coding standard enabled in the analysis (for instance, MISRA C: 2012, custom rules, and so on). Each chapter contains the following information:

- **Summary - Violations by File**: Graph showing each file with number of rule violations.
- **Summary - Violations by Rule**: Graph showing each rule with number of violations. If a rule is not enabled or not violated, it does not appear in the graph.
- **Summary for all Files**: Table showing each file with number of rule violations.
- **Summary for Enabled Guidelines** or **Summary for Enabled Rules**: Table showing each guideline or rule with number of violations.
- **Violations**: Tables listing each rule violation, along with information such as ID, function name, severity, status, and so on. One table is created per file.

An appendix lists the options used in the Polyspace analysis.

`Developer`

The report lists information useful to developers, including:

- Summary of verification results
- Code metrics summary
- Coding rules:
  - Summary of violations by file
  - List of violations
- Run-time errors:
  - List of proven run-time errors or red checks
  - List of unproven run-time errors or orange checks
  - List of unreachable procedures or gray checks
- Global variable usage in code. See "Global Variables" (Polyspace Code Prover).

The report also contains the Polyspace configuration settings and modifiable assumptions used in the analysis. If your project has source files with compilation errors, these files are also listed.

`DeveloperReview`

The report lists the same information as the `Developer` report. However, the reviewed results are sorted by severity and status, and unreviewed results are sorted by file location.

`Developer_withGreenChecks`

The report lists the same information as the `Developer` report. In addition, the report lists code proven to be error-free or green checks.

`Quality`

The report lists information useful to quality engineers, including:

- Summary of results
- Statistics about the code
- Graphs showing distributions of checks per file

The report also contains the Polyspace configuration settings and modifiable assumptions used in the analysis. If your project has source files with compilation errors, these files are also listed.

`VariableAccess`

The report displays the global variable access in your source code. The report first displays the number of global variables of each type. For information on the types, see "Global Variables" (Polyspace Code Prover). For each global variable, the report displays the following information:

- Variable name.

  The entry for each variable is denoted by |.
- Type of the variable.
- Number of read and write operations on the variable.
- Details of read and write operations. For each read or write operation, the table displays the following information:

  - File and function containing the operation in the form *`file_name.function_name`*.

    The entry for each read or write operation is denoted by `||`. Write operations are denoted by < and read operations by >.
  - Line and column number of the operation.

This report captures the information available on the **Variable Access** pane in the Polyspace user interface.

`CallHierarchy`

The report displays the call hierarchy in your source code. For each function call in your source code, the report displays the following information:

- Level of call hierarchy, where the function is called.

  Each level is denoted by |. If a function call appears in the table as `|||->` *`file_name.function_name`*, the function call occurs at the third level of the hierarchy. Beginning from `main` or an entry point, there are three function calls leading to the current call.
- File containing the function call.

  In addition, the line and column is also displayed.
- File containing the function definition.

  In addition, the line and column where the function definition begins is also displayed.

In addition, the report also displays uncalled functions.

This report captures the information available on the **Call Hierarchy** pane in the Polyspace user interface.

`SoftwareQualityObjectives`

The report lists information useful to quality engineers and available on the Polyspace Access interface, including:

- Information about whether the project satisfies quality objectives
- Metrics about the whole project. For each metric, the report lists the quality threshold and whether the metric satisfies this threshold.
- Coding rule violations in the project. For each rule, the report lists the number of violations justified and whether the justifications satisfy quality objectives.
- Definite as well as possible run-time errors in the project. For each type of run-time error, the report lists the number of errors justified and whether the justifications satisfy quality objectives.

The appendices contain further details of Polyspace configuration settings, code metrics, coding rule violations, and run-time errors.

This template is available only if you generate a report from results uploaded to the Polyspace Access web interface. See “Upload Results to Polyspace Access”. In each case, you have to set the objectives explicitly in the web interface and then generate the reports.

For more information on the predefined Software Quality Objectives, see “Evaluate Polyspace Code Prover Results Against Software Quality Objectives” (Polyspace Code Prover).

`SoftwareQualityObjectives_Summary`

The report contains the same information as the `SoftwareQualityObjectives` report. However, it does not have the supporting appendices with details of code metrics, coding rule violations and run-time errors.

This template is available only if you generate a report from results uploaded to the Polyspace Access web interface. See “Upload Results to Polyspace Access”. In each case, you have to set a quality objective level explicitly in the web interface and then generate the reports.

For more information on the predefined Software Quality Objectives, see “Evaluate Polyspace Code Prover Results Against Software Quality Objectives” (Polyspace Code Prover).

## Dependencies

In the user interface of the Polyspace desktop products, this option is enabled only if you select the `Generate report` option.

## Tips

- This option allows you to specify report generation before starting an analysis.

  To generate a report *after* an analysis is complete, in the user interface of the Polyspace desktop products, select **Reporting** > **Run Report**. Alternatively, at the command line, use the `polyspace-report-generator` command.

  After analysis, you can also export the result as a text file for further customization. Use the option `-generate-results-list-file` with the `polyspace-report-generator` command.
- In Bug Finder, the report does not contain the line or column number for a result. Use the report for archiving, gathering statistics and checking whether results have been reviewed and addressed (for certification purposes or otherwise). To review a result in your source code, use the Polyspace desktop interface, the Polyspace Access web interface, or your IDE if you are using a Polyspace plugin.
- If you use the `SoftwareQualityObjectives_Summary` and `SoftwareQualityObjectives` templates to generate reports, the pass/fail status is determined based on all results. For instance, if you use the level SQO-4 which sets a threshold of 60% on orange overflow checks, your project has a **FAIL** status if the percentage of green and justified orange overflow checks is less than 60% of *all green and orange overflow checks*.
- The first chapter of the reports contain a summary of the relevant results. You can enter a Pass/Fail status in that chapter for your project based on the summary. If you use the template `SoftwareQualityObjectives` or `SoftwareQualityObjectives_Summary`, the status is automatically assigned based on your objectives and the verification results. For more information

on enforcing objectives using Polyspace Access, see "Quality Objectives Dashboard in Polyspace Access".

## Command-Line Information

**Parameter:** `-report-template`
**Value:** Full path to *`template`*`.rpt`
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-report-template` *`polyspaceroot`*`\toolbox\polyspace\psrptgen\templates\bug_finder\BugFinder.rpt`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-report-template` *`polyspaceroot`*`\toolbox\polyspace\psrptgen\templates\Developer.rpt`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-report-template` *`polyspaceroot`*`\toolbox\polyspace\psrptgen\templates\bug_finder\BugFinder.rpt`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-report-template` *`polyspaceroot`*`\toolbox\polyspace\psrptgen\templates\Developer.rpt`

## See Also

`Generate report` | `Output format (-report-output-format)` | `polyspace-report-generator`

### Topics

"Specify Polyspace Analysis Options"
"Generate Reports from Polyspace Results"

# Output format (-report-output-format)

Specify output format of generated report

## Description

Specify output format of analysis report.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Reporting** node. See "Dependencies" on page 2-337 for other options you must also enable.

**Command line and options file**: Use the option `-report-output-format`. See "Command-Line Information" on page 2-337.

### Why Use This Option

Use this option to specify whether you want a report in PDF, HTML or another format.

## Settings

**Default:** `Word`

`HTML`

Generate report in `.html` format

`PDF`

Generate report in `.pdf` format

`Word`

Generate report in `.docx` format.

## Tips

- This option allows you to specify report generation before starting an analysis.

  To generate a report *after* an analysis is complete, in the user interface of the Polyspace desktop products, select **Reporting** > **Run Report**. Alternatively, at the command line, use the `polyspace-report-generator` command.

  After analysis, you can also export the result as a text file for further customization. Use the option `-generate-results-list-file` with the `polyspace-report-generator` command.
- If the table of contents or graphics in a `.docx` report appear outdated, select the content of the report and refresh the document. Use keyboard shortcuts **Ctrl+A** to select the content and **F9** to refresh it.

## Dependencies

In the user interface of the Polyspace desktop products, this option is enabled only if you select the `Generate report` option.

## Command-Line Information

**Parameter:** `-report-output-format`
**Value:** `html | pdf | word`
**Default:** `word`
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`file_name`* `-report-output-format pdf`
**Example (Code Prover):** `polyspace-code-prover -sources` *`file_name`* `-report-output-format pdf`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`file_name`* `-report-output-format pdf`
**Example (Code Prover Server):** `polyspace-code-prover-server -sources` *`file_name`* `-report-output-format pdf`

## See Also

`Generate report` | `Bug Finder and Code Prover report (-report-template)` | `polyspace-report-generator`

### Topics

"Specify Polyspace Analysis Options"
"Generate Reports from Polyspace Results"

# Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`)

Enable batch remote analysis

## Description

Specify that the analysis must be offloaded to a remote server.

To offload a Polyspace analysis, you need these products:

- A Polyspace product on the client side to submit an analysis job. Typically, you use a desktop product such as Polyspace Bug Finder to submit jobs, but you can also use a server product such as Polyspace Bug Finder Server to offload an analysis from one server to another.
- A Polyspace server product (Polyspace Bug Finder Server or Polyspace Code Prover Server) on the server side to run the analysis.
- MATLAB Parallel Server™ to hold jobs from multiple clients in queue and allocate the jobs as Polyspace Server instances become available.

For details, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Run Settings** node. You have separate options for a Bug Finder and a Code Prover analysis.

**Command line and options file**: Use the option `-batch`. See “Command-Line Information” on page 2-339.

### Why Use This Option

Use this option if you want the analysis to run on a remote cluster instead of your local desktop.

For instance, you can run remote analysis when:

- You want to shut down your local machine but not interrupt the analysis.
- You want to free execution time on your local machine.
- You want to transfer the analysis to a more powerful computer.

## Settings

☑ On

Run batch analysis on a remote computer. In this remote analysis mode, the analysis is queued on a cluster after the compilation phase. Therefore, on your local computer, after the analysis is queued:

- If you are running the analysis from the Polyspace user interface, you can close the user interface.

- If you are running the analysis from the command line, you can close the command-line window.

You can manage the queue from the Polyspace Job Monitor. To use the Polyspace Job Monitor:

- In the Polyspace user interface, select **Tools** > **Open Job Monitor**. See “Send Polyspace Analysis from Desktop to Remote Servers”.
- On the DOS or UNIX® command line, use the `polyspace-jobs-manager` command. For more information, see “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”.
- On the MATLAB command line, use the `polyspaceJobsManager` function.

**Note** If you use a third-party scheduler instead of the MATLAB Job Scheduler, disable the credentials check by adding the option `-no-credentials-check`. The credentials check performed in the product is compatible only with the MATLAB Job Scheduler and the check outputs an error if you use a different scheduler. In the Polyspace user interface, add this option to the `Other` field.

□ Off (default)

Do not run batch analysis on a remote computer.

## Dependencies

- If you use a Polyspace Server product to offload an analysis from one server to another, the offloading uses the MATLAB Job Scheduler that comes by default with MATLAB Parallel Server. You cannot use a third-party scheduler.

## Tips

- If you run a Code Prover analysis and set the **Verification Level** to `Source Compliance Checking`, the analysis takes place on your local machine even when you enable remote analysis. Therefore, if you run an analysis only up to this verification level, use your local machine.
- In the user interface of the Polyspace desktop products, the results are automatically downloaded after analysis.

  If you run an analysis at the command line, add the options `-wait -download` to automatically download the results after analysis. Otherwise, you can explicitly download the results later with the `polyspace-jobs-manager` command. See “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”.
- If you use a Polyspace Server product to offload an analysis from one server to another, the results are automatically downloaded after analysis.

## Command-Line Information

To run a remote analysis from the command line, use with the `-scheduler` option.
**Parameter:** `-batch`
**Value:** `-scheduler` *`host_name`* if you have not set the **Job scheduler host name** in the Polyspace user interface

**Default:** Off
**Example (Bug Finder):** `polyspace-bug-finder -batch -scheduler NodeHost` or `polyspace-bug-finder -batch -scheduler MJSName@NodeHost`
**Example (Code Prover):** `polyspace-code-prover -batch -scheduler NodeHost` or `polyspace-code-prover -batch -scheduler MJSName@NodeHost`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -batch -scheduler NodeHost`
**Example (Code Prover Server):** `polyspace-code-prover-server -batch -scheduler NodeHost`

## See Also

`-scheduler`

### Topics

"Install Products for Submitting Polyspace Analysis from Desktops to Remote Server"
"Specify Polyspace Analysis Options"
"Send Polyspace Analysis from Desktop to Remote Servers"
"Send Polyspace Analysis from Desktop to Remote Servers Using Scripts"
"Send Analysis from Client to Server"
"Offload Polyspace Analysis from Continuous Integration Server to Another Server"

# Use fast analysis mode for Bug Finder (`-fast-analysis`)

Run analysis using faster local mode

## Description

*This option affects a Bug Finder analysis only.*

Run analysis using faster local mode. The first run analyzes all files, but subsequent runs reanalyze only the files that you edited since the previous analysis.

Fast analysis mode is a faster way to analyze code for localized defects and coding standard violations. When you launch fast analysis, Bug Finder analyzes your code for a subset of defects and coding rules. These defects and coding standard violations are ones that can be found in the early stages of the analysis or can leverage archived information from a previous analysis. The analysis results are also comparatively easier to review and fix because most results can be understood by focusing on two or three lines of code (the line with the defect and one or two previous events).

Because of the simplified nature of the analysis, you might see significantly fewer defects in the fast analysis mode compared to a regular Bug Finder analysis.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is available on the **Run Settings** node.

**Command line and options file**: Use the option `-fast-analysis`. See “Command-Line Information” on page 2-343.

### Why Use This Option

If you use this option, you have to wait less for analysis results from your second analysis onwards. During development, you can frequently run analysis in fast mode and quickly fix new defects or coding standard violations.

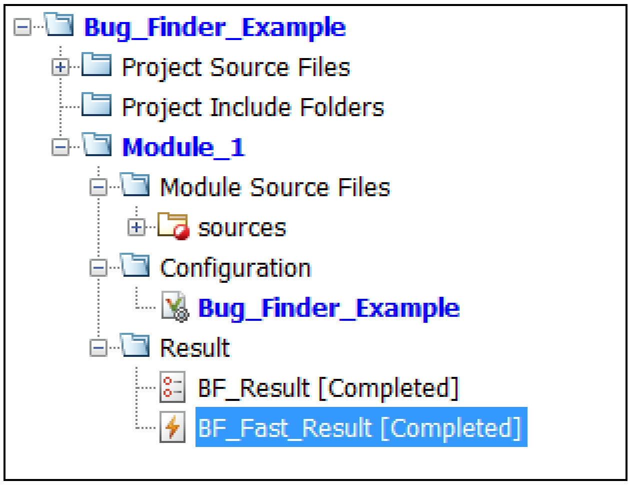
Polyspace produces results quickly because the analysis is localized. When you rerun in fast-analysis mode, Polyspace reanalyzes only those files that need to be reanalyzed, regenerating results even more quickly. These situations trigger a reanalysis.

| Situation | What Is Reanalyzed |
|---|---|
| You modified a source file. | Modified source file |
| You modified a header file. | Source files that include the modified header file (directly or indirectly) |
| You added or removed an analysis option. | All files |
| Previous fast-analysis results were not found.<br>For instance, you deleted the results folder. | All files |

| Situation | What Is Reanalyzed |
|---|---|
| You upgraded to a later release of Polyspace and ran the analysis. | All files<br>Comments from the previous analysis are retained and imported to the current analysis. |

For example, consider a Polyspace project with three `.c` files and you fix a bug in one of the files. When you rerun the analysis, Polyspace reanalyzes only the one file that you changed.

The results of fast analysis appear in a folder separate from the results of normal analysis.



## Settings

**Default:** □ Off

☑ On

Polyspace Bug Finder runs in fast-analysis mode. Polyspace analyzes code for only a subset of defects and coding standard violations. If you have enabled checking of defects or coding standard violations that are not supported by fast-analysis, your code is not checked for those results.

□ Off

Polyspace Bug Finder runs in the normal mode. Analysis checks for all selected defects, coding standard violations, and code metrics.

## Tips

### Comments Import

If you enter comments in your results, the comments are automatically imported to the next analysis in fast mode.

To import the comments from fast mode results to results of a regular Bug Finder analysis, do one of the following:

- Select **Tools** > **Import Comments**. Navigate to the sibling results folder `BF_Fast_Result` and import comments from the fast mode results.
- When reviewing results of fast mode, enter the comments directly into your code. If you run a regular analysis on this code, the comments are imported to your analysis results.

  For details on how to enter code comments, see “Annotate Code and Hide Known or Acceptable Results”.

### Fast Analysis Limitations

In fast analysis mode, you cannot perform these actions:

- You cannot create a new results folder for each run. Even if you choose to create a new result folder, each new run overwrites the previous one.
- You cannot specify constraints using the option `Constraint setup (-data-range-specifications)`.
- You cannot run the analysis on a remote cluster.

## Command-Line Information

**Parameter:** `-fast-analysis`
**Default:** Off
**Example (Bug Finder):** `polyspace-bug-finder -sources` *`filename`* `-fast-analysis`
**Example (Bug Finder Server):** `polyspace-bug-finder-server -sources` *`filename`* `-fast-analysis`

## See Also

**Topics**
“Bug Finder Results Found in Fast Analysis Mode”

# Command/script to apply after the end of the code verification (`-post-analysis-command`)

Specify command or script to be executed after analysis

## Description

Specify a command or script to be executed after the analysis.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Advanced Settings** node.

**Command line and options file**: Use the option `-post-analysis-command`. See “Command-Line Information” on page 2-346.

### Why Use This Option

Create scripts for tasks that you want performed after the Polyspace analysis.

For instance, you want to be notified by email that the Polyspace analysis is over. Create a script that sends an email and use this option to execute the script after the Polyspace analysis.

## Settings

**No Default**

Enter full path to the command or script, or click 📁 to navigate to the location of the command or script. After the analysis, this script is executed.

The script is executed in the Polyspace results folder. In your script, consider the results folder as the current folder for relative paths to other files.

For a Perl script, in Windows, specify the full path to the Perl executable followed by the full path to the script. For example, to specify a Perl script `send_email.pl` that sends an email once the analysis is over, enter *polyspaceroot*`\sys\perl\win32\bin\perl.exe <`*absolute_path*`>\send_email.pl`. Here, *polyspaceroot* is the location of the current Polyspace installation, such as `C:\Program Files\Polyspace\R2019a\`, and <*absolute_path*> is the location of the Perl script.

For example Perl scripts that send emails and other information on writing Perl scripts, see Perl documentation.

## Tips

### Running post analysis commands on the server

If you perform verification on a remote server, after verification, the software executes your command on the server, not on the client desktop. If your command executes a script, the script must be present on the server.

For instance, if you specify the command, `/local/utils/send_mail.sh`, the Shell script `send_email.sh` must be present on the server in `/local/utils/`. The software does not copy the script `send_email.sh` from your desktop to the server before executing the command. If the script is not present on the server, you encounter an error. Sometimes, there are multiple servers that the MATLAB Job Scheduler can run the verification on. Place the script on each of the servers because you do not control which server eventually runs your verification.

### Running post analysis commands in the Polyspace user interface

To test the use of this option, run the following Perl script from a folder containing a Polyspace project (`.psprj` file). The script parses the latest Polyspace log file in the folder `Module_1\CP_Result` and writes the current project name and date to a file `report.txt`. The file is saved in `Module_1\CP_Result`.

```
foreach my $file (`ls Module_1\\CP_Result\\Polyspace_*.log`) {
  open (FH, $file);

while ($line = <FH>) {
    if ($line =~ m/Ending at: (.*)/) {
      $date=$1;
    }
    if ($line =~ m/-prog=(.*)/) {
      $project=$1;
    }
  }
}

my $filename = 'report.txt';
open(my $fh, '>', $filename) or die "Could not open file '$filename' $!";

print $fh "date=$date\n";
print $fh "project=$project\n";

close $fh;
```

In Linux, you can specify the Perl script for this option.

In Windows, instead of specifying the Perl script directly, specify a `.bat` file that invokes Perl and runs this script. For instance, the `.bat` file can contain the following line (assuming that the `.bat` file and `.pl` file are in the Polyspace project folder). Depending on your MATLAB installation, change the path to `perl.exe` appropriately.

```
"C:\Program Files\MATLAB\R2018b\sys\perl\win32\bin\perl.exe" command.pl
```

Run Code Prover. Check that the folder `Module_1\CP_Result` contains the file `report.txt` with the project name and date.

## Command-Line Information

**Parameter:** `-post-analysis-command`
**Value:** Path to executable file or command in quotes
**No Default**
**Example in Linux (Bug Finder):** `polyspace-bug-finder -sources file_name -post-analysis-command `pwd`/send_email.pl`
**Example in Linux (Code Prover) :** `polyspace-code-prover -sources file_name -post-analysis-command `pwd`/send_email.pl`
**Example in Linux (Bug Finder Server):** `polyspace-bug-finder-server -sources file_name -post-analysis-command `pwd`/send_email.pl`
**Example in Linux (Code Prover Server):** `polyspace-code-prover-server -sources file_name -post-analysis-command `pwd`/send_email.pl`
**Example in Windows:** `polyspace-bug-finder -sources file_name -post-analysis-command "C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"`

Note that in Windows, you use the full path to the Perl executable.

## See Also

`Command/script to apply to preprocessed files (-post-preprocessing-command)`

### Topics

"Specify Polyspace Analysis Options"

# Other

Specify additional flags for analysis

## Description

*This option is useful only if you run an analysis in the user interface of the Polyspace desktop products.*

Enter command-line-style flags such as `-max-processes`.

### Set Option

**User interface** (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. You can enter multiple options in this field. If you enter the same option multiple times with different arguments, the analysis uses your last argument.

### Why Use This Option

Use this option to add nonofficial or command-line only options to the analyzer.

If you have to add several command line options, you can save them in a text file and specify the file using the option `-options-file`. You can reuse the options file across projects.

## Tip

Nonofficial options: In rare circumstances, to work around very specific issues, MathWorks Technical Support might provide you some undocumented options. If you are running verification from the user interface, you use the **Other** field in the **Configuration** pane to enter the options. Sometimes, the options and their arguments have to be preceded by extra flags. When providing you the option, Technical Support will let you know if the extra flags are required.
**Possible Flags:** `-extra-flags | -c-extra-flags | -cpp-extra-flags | -cfe-extra-flags | -il-extra-flags`
**Example (Bug Finder):** `polyspace-bug-finder -extra-flags` *`-option-name`* `-extra-flags` *`option_param`*
**Example (Code Prover):** `polyspace-code-prover -extra-flags` *`-option-name`* `-extra-flags` *`option_param`*
**Example (Bug Finder Server):** `polyspace-bug-finder-server -extra-flags` *`-option-name`* `-extra-flags` *`option_param`*
**Example (Code Prover Server):** `polyspace-code-prover-server -extra-flags` *`-option-name`* `-extra-flags` *`option_param`*

3

# Analysis Options, Command-Line Only

# -asm-begin -asm-end

Exclude compiler-specific `asm` functions from analysis

## Syntax

`-asm-begin "`*`mark1[,mark2,...]`*`" -asm-end "`*`mark1[,mark2,...]`*`"`

## Description

`-asm-begin "`*`mark1[,mark2,...]`*`" -asm-end "`*`mark1[,mark2,...]`*`"` excludes compiler-specific assembly language source code functions from the analysis. You must use these two options together.

Polyspace recognizes most inline assemblers by default. Use the option only if compilation errors occur due to introduction of assembly code. For more information, see “Code Prover Assumptions About Assembly Code” (Polyspace Code Prover).

Mark the offending code block by two `#pragma` directives, one at the beginning of the assembly code and one at the end. In the command usage, give these marks in the same order for `-asm-begin` as they are for `-asm-end`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

A block of code is delimited by `#pragma start1` and `#pragma end1`. These names must be in the same order for their respective options. Either:

```
-asm-begin "start1" -asm-end "end1"
```

or

```
-asm-begin "mark1,...markN,start1" -asm-end "mark1,...markN,end1"
```

The following example marks two functions for exclusion, `foo_1` and `foo_2`.

Code:

```
#pragma asm_begin_foo
int foo(void) { /* asm code to be ignored by Polyspace */ }
#pragma asm_end_foo

#pragma asm_begin_bar
void bar(void) { /* asm code to be ignored by Polyspace */ }
#pragma asm_end_bar
```

Polyspace Command:

- Bug Finder:

```
polyspace-bug-finder -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
          -asm-end "asm_end_foo,asm_end_bar"
```

- Code Prover:

```
polyspace-code-prover -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
         -asm-end "asm_end_foo,asm_end_bar"
```

- Bug Finder Server:

```
polyspace-bug-finder-server -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
          -asm-end "asm_end_foo,asm_end_bar"
```

- Code Prover Server:

```
polyspace-code-prover-server -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
         -asm-end "asm_end_foo,asm_end_bar"
```

`asm_begin_foo` and `asm_begin_bar` mark the beginning of the assembly source code sections to be ignored. `asm_end_foo` and `asm_end_bar` mark the end of those respective sections.

## Tips

If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

## See Also

**Topics**
"Specify Polyspace Analysis Options"

# -author

Specify project author

## Syntax

```
-author "value"
```

## Description

`-author "value"` assigns an author to the Polyspace project. The name appears as the project owner on generated reports.

The default value is the user name of the current user, given by the DOS or UNIX command `whoami`.

In the user interface of the Polyspace desktop products, select to specify the Project name, Version, and Author parameters in the **Polyspace Project – Properties** dialog box.

## Examples

Assign a project author to your Polyspace Project.

- Bug Finder:

  ```
  polyspace-bug-finder -author "John Smith"
  ```
- Code Prover:

  ```
  polyspace-code-prover -author "John Smith"
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -author "John Smith"
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -author "John Smith"
  ```

## Tips

This option is not required for a Polyspace as You Code analysis.

## See Also

`-prog` | `-date`

### Topics

“Specify Polyspace Analysis Options”

# -c

Specify path of license file used by the product

## Syntax

`-c licensePath`

## Description

`-c licensePath` specifies the path of the license file that the product uses. When you specify this option, Polyspace looks for only this file to use as a license. Other possible locations such as environment variables `MLM_LICENSE_FILE` and `LM_LICENSE_FILE`, or the `licenses` folder under the Polyspace installation folder are ignored.

By default, Polyspace looks for a license in environment variables, registry keys, user profiles, and the `licenses` folder, in that order. For more details, see What is the license file search path.

Use this option to, for instance, troubleshoot your installation if Polyspace cannot find a license in the default locations, or to use a different license if the currently checked out license has no more seats available.

**Note** When you use this option to specify a Polyspace license, you can specify only a license path. The use of the port and hostname of the license server (For example `-c 27000@lisenseServerHostname`) is not supported.

## Examples

Use license `C:\ps_license\license.lic` when you start the desktop interface or a Polyspace analysis.

- Polyspace desktop interface

  ```
  polypace -c C:\ps_license\license.lic
  ```
- Bug Finder analysis:

  ```
  polypace-bug-finder -c C:\ps_license\license.lic -sources...
  ```
- Code Prover analysis:

  ```
  polypace-code-prover -c C:\ps_license\license.lic -sources...
  ```
- Bug Finder Server analysis:

  ```
  polypace-bug-finder-server -c C:\ps_license\license.lic -sources...
  ```
- Code Prover Server analysis:

  ```
  polypace-code-prover-server -c C:\ps_license\license.lic -sources...
  ```

## See Also

### Topics

“Borrow a Polyspace License”
“Specify Polyspace Analysis Options”

# -checkers-activation-file

Enable a custom combination of defects and coding rules for a Polyspace Bug Finder analysis

## Syntax

`-checkers-activation-file` *`checkersFile`*`.xml`

## Description

`-checkers-activation-file` *`checkersFile`*`.xml` enables specific defect and coding standards checkers for a Bug Finder analysis. You can create the XML file for checkers activation in one of the following ways:

- Select the checkers in a graphical user interface and save your selection in a file *`checkersFile`*`.xml`. See “Create Checkers Activation File in Graphical User Interface” on page 3-7.
- Open an existing checkers activation XML file in a text editor and turn on checkers. See “Update Checkers Activation File in Text Editor” on page 3-8.

You can specify an absolute path to the XML file or a path relative to the location from which you run the `polyspace-bug-finder` command.

If you use an activation file generated by using a previous version of Polyspace, previously unimplemented checkers might become implemented. Polyspace warns if the file XML file lists an implemented checker as `'notimplemented'`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

### Create Checkers Activation File in Graphical User Interface

Create a checkers activation file by using the Checkers selection interface. This file is a reusable selection of Bug Finder defects and coding rules that you can use with the option `-checkers-activation-file` in a Bug Finder analysis.

1. Open the Checkers selection interface. At the command line, enter:

   ```
   polyspace-checkers-selection
   ```

   Alternatively, navigate to *`polyspaceroot`*`\polyspace\bin` and launch the binary `polyspace-checkers-selection`. Here, *`polyspaceroot`* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace Server\R2023a`.
2. In the Checkers selection interface, select **New**. Select defect checkers and coding rules that you want enabled. Click **Save Changes** to save your selection as an XML file, for instance, *`checkers.xml`*.
3. When running a Bug Finder analysis, specify this XML file as the argument for the option `-checkers-activation-file`.

```
polyspace-bug-finder-access -sources src.c -checkers-activation-file checkers.xml
```

### Update Checkers Activation File in Text Editor

In some cases, you might not have access to the Checkers Selection UI. For instance, you might be using Polyspace in a headless terminal. In this case, create the checkers activation XML file by using the template files in `*polyspaceroot*\help\toolbox\bugfinder\examples\checkers_activation_XML`.

To activate Bug Finder defects or checkers from one of the coding standards, make a writable copy of the corresponding templates. Locate the checker by finding the node that has the correct `check id` attribute. Activate the checker by setting the `state` attribute to `"on"`.

Alternatively, to make a custom XML that activates various coding standards and Bug Finder defects, copy the relevant entries from the corresponding coding standard and defect templates into the `blank.xml` template in `*polyspaceserverroot*\help\toolbox\bugfinder\examples\checkers_activation_XML`. A checkers activation XML file that activates the Bug Finder defect `DATA_RACE` and CERT C coding rule `CON01-C` resembles this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<polyspace_checkers_selection xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="selection_schema.xsd" revision="2.0">
  <standard name="Bug Finder Findings">
      <section name="Concurrency">
        <check id="DATA_RACE" state="on"/>
      </section>
  </standard>

  <standard name="SEI CERT C">
      <section name="Concurrency (CON)">
        <check id="CON01-C" state="on"/>
      </section>
    </standard>
</polyspace_checkers_selection>
```

For a full list of coding rule check IDs, defect checker short names, and section names, see:

- “Defects”
- “AUTOSAR C++14 Rules”
- “CERT C Rules and Recommendations”
- “CERT C++ Rules”
- “ISO/IEC TS 17961 Rules”
- “Custom Coding Rules”
- “JSF C++ Rules”
- “MISRA C:2004 Rules”
- “MISRA C:2012 Directives and Rules”
- “MISRA C++:2008 Rules”
- “Guidelines”

**Note** The XML format of the checker configuration file might change in future releases.

## Tips

- If you use Polyspace as You Code extensions in IDEs, this option is implemented through the IDE extension setting. You do not have to use this option explicitly. If you want to explicitly use this option, enter the option in an analysis options file. See options file.
- To create a custom coding standard classification, or to tag coding rule checkers of interest, enter text in the **Comment** column of the **Checkers selection** window. Polyspace displays that text in the **Results Details** pane and in the **Detail** column of the **Results List** (if available) when you review results in the desktop interface, in Polyspace Access, or in the Polyspace as You Code plugins.

## See Also

`Find defects (-checkers)`

### Topics

"Configure Checkers for Polyspace as You Code at the Command Line"

# -classification

Control precisely which files to include in Polyspace Bug Finder analysis and how to analyze them

## Syntax

`-classification` *`file`*

## Description

`-classification` *`file`* specifies which files to include in a Polyspace Bug Finder analysis and how to analyze them. Here, *`file`* is an XML file that enumerates file sets with specific behaviors.

For instance, you can:

- Skip the definitions of function bodies in third-party libraries.
- Force analysis of all functions in files that you own.

If you run a verification:

- At the command line, specify the absolute path to the XML files or path relative to the folder from which you run the command.
- In the user interface (desktop products only), in the **Other** field, specify the option along with an absolute path to the XML or the path relative to the location of your Polyspace project. See `Other`.

A report generated from the analysis results only show the use of this option and not the details of which behaviors were associated with code elements.

A sample template file `classification-template.xml` shows the XML syntax. The file is in *`polyspaceroot`*`\polyspace\verifier\cxx\` where *`polyspaceroot`* is the Polyspace installation folder.

## Examples

### Excluding Generated Files

Suppose your source code consists of the following files. Both `.c` files contain a comparison that incorrectly uses an = operator instead of an == operator.

- `src\utils.c`:

  ```
  #include "header.h"

  int getEvenElement(int *arr, int size, int idx) {
      int isIndexEven = isEven(idx);
      if(isIndexEven = 0 && idx < size) // Invalid use of = operator
          return arr[idx];
      else
          return -1;
  }
  ```
- `src\utils_generated.c`:

```
#include "header.h"

int isEven(int arg) {
    int rem = arg%2;
    if(rem = 0) { // Invalid use of = operator
        return 1;
    }
    else {
        return 0;
    }
}
```

- `inc\header.h`:

  ```
  int isEven(int);
  ```

Run the default Bug Finder analysis:

```
polyspace-bug-finder -sources src\utils.c,src\utils_generated.c -I inc\
```

You see an `Invalid use of = operator` defect in both `.c` files.

Suppose your generated files follow a naming convention and end with `_generated`. To exclude these files from analysis:

1. Write a classification file that defines a set of files to exclude from analysis.

   ```
   <?xml version="1.0" encoding="UTF-8"?>

   <specification>
       <classification product="bug-finder">
          <fileset name="generated code">
               <files-in-set>
                   <file-pattern>src/**/*_generated.c</file-pattern>
               </files-in-set>
               <behaviors>
                   <do-not-analyze-functions>
                       <function-pattern>*</function-pattern>
                   </do-not-analyze-functions>
                   <show-results value="false"/>
               </behaviors>
           </fileset>
       </classification>
   </specification>
   ```

2. Save the classification file as `exclude_generated_files.xml`. Specify this classification file with the option `-classification`:

   ```
   polyspace-bug-finder -sources src\file.c,src\file_generated.c -I inc\ -classification exclude
   ```

   You no longer see the defect in the file `utils_generated.c`.

## Version History

**Introduced in R2023a**

## See Also

**Topics**
“Specify Polyspace Analysis Options”
“Classify Project Files Into File Sets for Precise Control of Bug Finder Analysis”
“Select Files for Polyspace Analysis Using Pattern Matching”

# -code-behavior-specifications

Associate behaviors with code elements such as functions

## Syntax

`-code-behavior-specifications *file*`

## Description

`-code-behavior-specifications` *`file`* allows you to associate certain behaviors with elements of your code and modify the results of checks on those elements. Here, *`file`* associates specific behaviors to code elements such as functions, and can be in one of two formats:

- An XML file.

  You can only specify a single XML file as argument. A sample template file `code-behavior-specifications-template.xml` shows the XML syntax. The file is in *`polyspaceroot`*`\polyspace\verifier\cxx\` where *`polyspaceroot`* is the Polyspace installation folder.
- A Datalog (`.dl`) file.

  You can specify multiple Datalog files as comma-separated arguments.

You can specify an absolute path to the XML or Datalog file, or a path relative to the location from which you run the `polyspace-bug-finder` or `polyspace-code-prover` command.

The XML and Datalog format support different code behaviors. For more information, see:

- “Modify Bug Finder Checkers Through Code Behavior Specifications”
- “Modify Code Prover Run-Time Checks Through Code Behavior Specifications” (Polyspace Code Prover)

For instance, you can:

- Map your library functions to corresponding standard functions that Polyspace recognizes. Mapping to standard library functions can help with precision improvement or automatic detection of new threads.
- Specify that a function has a special behavior or must be subjected to special checks.

  For instance, you can specify that a function must only take addresses of initialized variables as arguments, or that a function must not be used altogether.

You can specify this option in the user interface of the Polyspace desktop products, at the command line or in IDE-s:

- If you run verification from the command line, specify the absolute path to the XML files or path relative to the folder from which you run the command.
- If you run verification from the user interface (desktop products only), in the **Other** field, specify the option along with an absolute path to the XML or the path relative to the location of your Polyspace project. See `Other`.

- If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

A report generated from the analysis results only show the use of this option and not the details of which behaviors were associated with code elements.

## Examples

### Specify Mapping to Standard Function

You can adapt the sample mapping XML file provided with your Polyspace installation and map your function to a standard function.

Suppose the default verification produces an orange `User assertion` check on this code:

```
double x = acos32(1.0);
assert(x <= 2.0);
```

Suppose you know that the function `acos32` behaves like the function `acos` and the return value is 0. You expect the check on the `assert` statement to be green. However, the verification considers that `acos32` returns any value in the range of type `double` because `acos32` is not precisely analyzed. The check is orange. To map your function `acos32` to `acos`:

1. Copy the file `code-behavior-specifications-template.xml` from *`polyspaceroot`* `\polyspace\verifier\cxx\` to another location, for instance, `"C:\Polyspace_projects\Common\Config_files"`. Change the write permissions on the file.
2. To map your function to a standard function, modify the contents of the XML file. To map your function `acos32` to the standard library function `acos`, change the following code:

   ```
   <function name="my_lib_cos" std="acos"> </function>
   ```

   To:

   ```
   <function name="acos32" std="acos"> </function>
   ```
3. Specify the location of the file for verification:

   - Code Prover:

     ```
     polyspace-code-prover -code-behavior-specifications
       "C:\Polyspace_projects\Common\Config_files
        \code-behavior-specifications-template.xml"
     ```
   - Code Prover Server:

     ```
     polyspace-code-prover-server -code-behavior-specifications
       "C:\Polyspace_projects\Common\Config_files
        \code-behavior-specifications-template.xml"
     ```

### Specify Mapping to Standard Function with Argument Remapping

Sometimes, the arguments of your function do not map one-to-one with arguments of the standard function. In those cases, remap your function argument to the standard function argument. For instance:

- `__ps_lookup_table_clip`:

This function specific to Polyspace takes only a look-up table array as argument and returns values within the range of the look-up table. Your look-up table function might have additional arguments besides the look-up table array itself. In this case, use argument remapping to specify which argument of your function is the look-up table array.

For instance, suppose a function `my_lookup_table` has the following declaration:

```
double my_lookup_table(double u0, const real_T *table,
                                     const double *bp0);
```

The second argument of your function `my_lookup_table` is the look-up table array. In the file `code-behavior-specifications-template.xml`, add this code:

```
<function name="my_lookup_table" std="__ps_lookup_table_clip">
    <mapping std_arg="1" arg="2"></mapping>
</function>
```

When you call the function:

```
res = my_lookup_table(u, table10, bp);
```

The verification interprets the call as:

```
res =__ps_lookup_table_clip(table10);
```

The verification assumes that the value of `res` lies within the range of values in `table10`.

- `__ps_meminit`:

  This function specific to Polyspace takes a memory address as the first argument and a number of bytes as the second argument. The function assumes that the bytes in memory starting from the memory address are initialized with a valid value. Your memory initialization function might have additional arguments. In this case, use argument remapping to specify which argument of your function is the starting address and which argument is the number of bytes.

  For instance, suppose a function `my_meminit` has the following declaration:

```
     void my_meminit(enum InitKind k, void* dest, int is_aligned,
                           unsigned int size);
```

  The second argument of your function is the starting address, and the fourth argument is the number of bytes. In the file `code-behavior-specifications-template.xml`, add this code:

```
<function name="my_meminit" std="__ps_meminit">
    <mapping std_arg="1" arg="2"></mapping>
    <mapping std_arg="2" arg="4"></mapping>
</function>
```

  When you call the function:

```
my_meminit(INIT_START_BY_END, &buffer, 0, sizeof(buffer));
```

  The verification interprets the call as:

```
__ps_meminit(&buffer, sizeof(buffer));
```

  The verification assumes that `sizeof(buffer)` number of bytes starting from `&buffer` are initialized.

- memset: Variable number of arguments.

  If your function has variable number of arguments, you cannot map it directly to a standard function without explicit argument remapping. For instance, say your function is declared as:

  ```
  void* my_memset(void*, int, size_t, ...)
  ```

  To map the function to the memset function, use the following mapping:

  ```
  <function name="my_memset" std="memset">
      <mapping std_arg="1" arg="1"></mapping>
      <mapping std_arg="2" arg="2"></mapping>
      <mapping std_arg="3" arg="3"></mapping>
  </function>
  ```

#### Effect of Mapping on Precision

These examples show the result of mapping certain functions to standard functions:

- my_acos → acos:

  If you use the mapping, the User assertion check turns green. The verification assumes that the return value of my_acos is 0.

  - *Before mapping*:

    ```
    double x = my_acos(1.0);
    assert(x <= 2.0);
    ```
  - *Mapping specification*:

    ```
    <function name="my_acos" std="acos">
    </function>
    ```
  - *After mapping*:

    ```
    double x = my_acos(1.0);
    assert(x <= 2.0);
    ```
- my_sqrt → sqrt:

  If you use the mapping, the Invalid use of standard library routine check turns red. Otherwise, the verification does not check whether the argument of my_sqrt is nonnegative.

  - *Before mapping*:

    ```
    res = my_sqrt(-1.0);
    ```
  - *Mapping specification*:

    ```
    <function name="my_sqrt" std="sqrt">
    </function>
    ```
  - *After mapping*:

    ```
    res = my_sqrt(-1.0);
    ```
- my_lookup_table (argument 2) →__ps_lookup_table_clip (argument 1):

  If you use the mapping, the User assertion check turns green. The verification assumes that the return value of my_lookup_table is within the range of the look-up table array table.

  - *Before mapping*:

    ```
    double table[3] = {1.1, 2.2, 3.3}
    .
    .
    double res = my_lookup_table(u, table, bp);
    assert(res >= 1.1 && res <= 3.3);
    ```

  - *Mapping specification*:

    ```
    <function name="my_lookup_table" std="__ps_lookup_table_clip">
        <mapping std_arg="1" arg="2"></mapping>
    </function>
    ```

  - *After mapping*:

    ```
    double table[3] = {1.1, 2.2, 3.3}
    .
    .
    res_real = my_lookup_table(u, table9, bp);
    assert(res_real >= 1.1 && res_real <= 3.3);
    ```

- `my_meminit` → `__ps_meminit`:

  If you use the mapping, the `Non-initialized local variable` check turns green. The verification assumes that all fields of the structure `x` are initialized with valid values.

  - *Before mapping*:

    ```
    struct X {
      int field1;
      int field2;
    };
    .
    .
    struct X x;
    my_meminit(&x, sizeof(struct X));
    return x.field1;
    ```

  - *Mapping specification*:

    ```
    <function name="my_meminit" std="__ps_meminit">
         <mapping std_arg="1" arg="1"></mapping>
         <mapping std_arg="2" arg="2"></mapping>
    </function>
    ```

  - *After mapping*:

    ```
    struct X {
      int field1 ;
      int field2 ;
    };
    .
    .
    struct X x;
    my_meminit(&x, sizeof(struct X));
    return x.field1;
    ```

- `my_meminit` → `__ps_meminit`:

  If you use the mapping, the `Non-initialized local variable` check turns red. The verification assumes that only the field `field1` of the structure `x` is initialized with valid values.

- *Before mapping*:

  ```
  struct X {
    int field1 ;
    int field2 ;
  };
  .
  .
  struct X x;
  my_meminit(&x, sizeof(int));
  return x.field2;
  ```
- *Mapping specification*:

  ```
  <function name="my_meminit" std="__ps_meminit">
  </function>
  ```
- *After mapping*:

  ```
  struct X {
    int field1 ;
    int field2 ;
  };
  .
  .
  struct X x;
  my_meminit(&x, sizeof(int));
  return x.field2;
  ```

## Limitations

You can map your custom functions to similar standard library functions using the option `-code-behavior-specifications`, subject to the following constraints:

- Your custom function must have the same number of arguments as the standard library function or more. Make sure that every argument of the standard library function is mapped to an argument of the custom function.
- The mapped function arguments must have compatible data types. Likewise, the custom function must have a return type that is compatible with the standard library function. For instance:
  - An integer type (`char`, `int`, etc.) is not compatible with a floating point type (`float`, `double`, etc.)
  - A fundamental type is not compatible with a structure or enumeration.
  - A pointer type is not compatible with a non-pointer type.

# Version History

**Introduced in R2016b**

## See Also

### Topics

“Specify Polyspace Analysis Options”

# -consider-analysis-perimeter-as-trust-boundary

Consider the analysis perimeter as trust boundary so that data coming from outside the current analysis perimeter is considered tainted

## Syntax

`-checkers tainted_data -consider-analysis-perimeter-as-trust-boundary`

## Description

`-checkers tainted_data -consider-analysis-perimeter-as-trust-boundary` modifies the behavior of the tainted data defects so that data originating from outside the analysis perimeter is considered tainted. For instance, if you are analyzing a single file, then any data that originates outside this file is considered tainted.

By default, these data are considered tainted:

- Objects declared or defined as `volatile`
- External data such as user input, hardware data, network data, and environment variable

See “Sources of Tainted Data”.

If you specify the option `-consider-analysis-perimeter-as-trust-boundary` along with the option `-checkers tainted_data`, data that does not originate in the current scope of Polyspace analysis is considered tainted. Such data might include:

- Formal parameters of externally visible function that do not have a visible caller.
- Return values of stubbed functions.
- Global variables external to the unit.

If you do not trust data that originates from an external module, use this option to detect operations that are vulnerable to this tainted data.

For a list of tainted data checkers that are impacted by this option, see “Polyspace Tainted Data Checkers”.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Example

Consider this code:

```
#include<stdio.h>
double taintedloopboundary(int num, int denum) {
    int count;
    scanf("%d", &count);
    for (int i=0; i<count; ++i) {
        num = num/denum;
    }
    return num;
}
```

The example shows two cases of tainted data: one detected by default and one using this option.

- The variable `count` is obtained from the user. If you perform the default tainted data analysis by using the command:

  ```
  polyspace-bug-finder -checkers tainted_data
  ```

  Polyspace flags the tainted loop variable.
- The variables `num` and `denum` are not defined in the current module. If you modify the tainted data analysis by using the command:

  ```
  polyspace-bug-finder -checkers tainted_data -consider-analysis-perimeter-as-trust-boundary
  ```

  Polyspace flags the division operation between `num` and `denum`.

## Tips

This option is not useful in a Polyspace as You Code analysis.

# Version History

**Introduced in R2020b**

## See Also

**Topics**
“Tainted Data Defects”
“Sources of Tainting in a Polyspace Analysis”
“Modify Default Behavior of Bug Finder Checkers”

# -custom-target

Create a custom target processor with specific data type sizes

## Syntax

`-custom-target target_sizes`

## Description

`-custom-target` *`target_sizes`* defines a custom target processor for the Polyspace analysis. The target processor definition includes sizes in bytes of fundamental data types, signedness of plain `char`, alignment of structures and underlying types of standard `typedef`-s such as `size_t`, `ptrdiff_t` and `wchar_t`.

*`target_sizes`* is a comma-separated list specifying these values. From left to right, the values are the following. If a data type is not supported, -1 is used for its size.

| Specification | Possible Values |
|---|---|
| Whether plain `char` is signed | `true` or `false` |
| Size of `char` in bits<br>Other sizes are in bytes. | Number |
| Size of `short` | Number |
| Size of `int` | Number |
| Size of `short long` | Number |
| Size of `long` | Number |
| Size of `long long` | Number |
| Size of `float` | Number |
| Size of `double` | Number |
| Size of `long double` | Number |
| Size of pointer | Number |
| Minimum alignment of all integer types | Number |
| Minimum alignment of variables of type `struct` or `union` | Number |
| Endianness | `little` or `big` |
| Underlying type of `size_t` | `unknown`, `unsigned_char`, `unsigned_short`, `unsigned_int`, `unsigned_long`, or `unsigned_long_long` |
| Underlying type of `ptrdiff_t` | `unknown`, `signed_char`, `short`, `int`, `long`, or `long_long` |
| Underlying type of `wchar_t` | `unknown`, `short`, `unsigned_short`, `int`, `unsigned_int`, `long`, or `unsigned_long` |

Typically, this option is used when the polyspace-configure command creates an options file for the subsequent Polyspace analysis. However, you can directly enter this option when manually writing options files. This option is useful in situations where your target specifications are not covered by one of the predefined target processors. See Target processor type (-target).

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See Other.

## Examples

An usage of the option looks like this:

```
-custom-target false,8,2,4,-1,4,8,4,8,8,4,8,1,little,unsigned_int,int,unsigned_int
```

The option argument translates to the following target specification.

| Specification | Possible Values |
|---|---|
| Whether plain char is signed | false |
| Size of char | 8 bits |
| Size of short | 2 bytes |
| Size of int | 4 bytes |
| Size of short long | short long is not supported. |
| Size of long | 4 bytes |
| Size of long long | 8 bytes |
| Size of float | 4 bytes |
| Size of double | 8 bytes |
| Size of long double | 8 bytes |
| Size of pointer | 4 bytes |
| Maximum alignment of all integer types | 8 bytes |
| Maximum alignment of variables of type struct or union | 1 byte |
| Endianness | little |
| Underlying type of size_t | unsigned_int |
| Underlying type of ptrdiff_t | int |
| Underlying type of wchar_t | unsigned_int |

## Tips

- If your configuration uses both -custom-target and Target processor type (-target) to specify targets, the analysis uses the target that you specify with -custom-target.
- If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

## See Also

Target processor type (-target) | Generic target options

**Topics**

"Specify Polyspace Analysis Options"

# -date

Specify date of analysis

## Syntax

`-date "*date*"`

## Description

`-date "date"` specifies the date stamp for the analysis in the format `dd/mm/yyyy`. By default the value is the date the analysis starts.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

Assign a date to your Polyspace Project:

- Bug Finder:

  ```
  polyspace-bug-finder -date "15/03/2012"
  ```
- Code Prover:

  ```
  polyspace-code-prover -date "15/03/2012"
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -date "15/03/2012"
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -date "15/03/2012"
  ```

## Tips

This option is not required for a Polyspace as You Code analysis.

## See Also

`-author`

### Topics

“Specify Polyspace Analysis Options”

# -detect-atomic-data-race

Enable detecting data race with atomic operation

## Syntax

`-detect-atomic-data-race`

## Description

`-detect-atomic-data-race` specifies that the analysis must check for data races in atomic operations..

By default, Polyspace does not check for data races in atomic operations. Bug Finder considers an operation as atomic if it can be performed in one machine instruction. For instance:

- The operation:

  ```
  int var = 0;
  ```

  can be performed in one machine instruction on targets where the size of `int` is less than the word size or the pointer size on the target.
- The operation:

  ```
  MYREG = (u32dma0_chbit << 8UL) | u32dma0_chbit;
  ```

  takes more than one cycle to be performed and is nonatomic.

See “Define Atomic Operations in Multitasking Code”. If you do not want to use this definition of atomic operations and want to check every operation for data races, specify the option `-detect-atomic-data-race`.

If you run verification from the user interface (desktop products only), specify the option in the **Other** field. See `Other`.

## Examples

In the code example, the write operation `var=1;` in task `task1` executes concurrently with the read operation `local_var=var;` in task `task2`. By default, Polyspace assumes that the write operation is atomic. Based on your environment, you might want to extend the `data race` checker to include the write operation.

### Code Example

```
#include<stdio.h>

int var;

void begin_critical_section(void);
void end_critical_section(void);
```

```
void task1(void) {
    var = 1;
}

void task2(void) {
    int local_var;
    local_var = var;
    printf("%d", local_var);
}

void task3(void) {
    begin_critical_section();
    /* Operations in task3 */
    end_critical_section();
}
```

To check the atomic operations in this example for data races, at the command line, specify these options:

```
 polyspace-bug-finder -sources SRC -checkers DATA_RACE
   -entry-points task1,task2,task3
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
   -detect-atomic-data-race
```

In the Polyspace user interface, use the options listed in this table.

| Option | Specification | |
|---|---|---|
| Other | -detect-atomic-data-race | |
| Configure multitasking manually | ☑ | |
| Tasks (-entry-points) | task1<br>task2<br>task3 | |
| Critical section details (-critical-section-begin -critical-section-end) | **Starting routine** | **Ending routine** |
| | begin_critical_section | end_critical_section |

## Dependencies

- To detect data races, you have to configure the concurrency options:

  - `Tasks (-entry-points)`
  - `Critical section details (-critical-section-begin -critical-section-end)`

  See “Multitasking”. If you specify the option `-detect-atomic-data-race` without configuring the concurrency options, Polyspace emits a warning.
- This option extends the following checkers and is not useful unless you enable at least one of the checkers:

- `Data race`
- `CERT C: Rule CON43-C`
- `CERT C++: CON43-C`

## Tips

This option is not useful in a Polyspace as You Code analysis.

## See Also

**Topics**
"Extend Data Race Checkers to Atomic Operations"

# -disable-concurrency-detection

Disable automatic detection of multitasking functions

## Syntax

`-disable-concurrency-detection`

## Description

`-disable-concurrency-detection` disables the automatic concurrency detection for supported multitasking functions.

Polyspace can automatically detect multitasking functions from several libraries, including these:

- POSIX
- VxWorks
- Windows
- µC/OS II
- C++11
- C11

See “Multitasking Routines that Polyspace Can Detect”.

Automatic detection of these functions might find too many tasks and slow down a Polyspace analysis. In such cases, use this option to disable automatic detection. Instead, manually configure your multitasking analysis by using analysis options such as `Tasks (-entry-points)`, `Critical section details (-critical-section-begin -critical-section-end)`, `Temporally exclusive tasks (-temporal-exclusions-file)`. See “Configuring Polyspace Multitasking Analysis Manually”.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

Disable automatic detection of multitasking functions by using this option. Then, manually configure the multitasking analysis:

```
polyspace-bug-finder -checkers concurrency -disable-concurrency-detection
-entry-points foo1,foo2,foo3
```

## Tips

This option is not useful in a Polyspace as You Code analysis.

## See Also

`Tasks (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

### Topics

“Auto-Detection of Thread Creation and Critical Section in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”

# -doc | -documentation

Display Polyspace documentation in help browser

## Syntax

```
-doc
-documentation
```

## Description

`-doc` and `-documentation` opens the Polyspace web documentation in your default web browser. You can see information such as getting started, workflows and reference pages for commands and analysis options. You can also search through the documentation.

## Examples

Display Polyspace documentation in a help browser:

- Bug Finder:

  ```
  polyspace-bug-finder -doc
  polyspace-bug-finder -documentation
  ```
- Code Prover:

  ```
  polyspace-code-prover -doc
  polyspace-code-prover -documentation
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -doc
  polyspace-bug-finder-server -documentation
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -doc
  polyspace-code-prover-server -documentation
  ```

## See Also

`-h[elp]`

# -dump-preprocessing-info

Show all macros implicitly defined during a particular analysis

## Syntax

`-dump-preprocessing-info`

## Description

`-dump-preprocessing-info` prints all the macros implicitly defined (or undefined) during a particular Polyspace analysis. The macro definitions come from:

- Your specification for the option `Compiler (-compiler)`

  Polyspace emulates a compiler by defining the compiler-specific macros.
- Macros defined (or undefined) in the Polyspace implementation of Standard Library headers
- Macros that you explicitly define (or undefine) using the options `Preprocessor definitions (-D)` and `Disabled preprocessor definitions (-U)`

Use this option only if you want to know how Polyspace defines a specific macro. In case you want to use a different definition for the macro, you can then override the current definition.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`. On the **Output Summary** pane, you can see each macro definition on a separate line. You can search for the macro name in the user interface and click the line with the macro name to see further details in the **Detail** pane.

## Examples

Suppose that you use the ARM v6 compiler for building your source code. For the Polyspace analysis, you use the value `armclang` for the option `Compiler (-compiler)`. Suppose that you want to know what Polyspace uses as definition for the macro `__ARM_ARCH`.

1. Enter the following command and pipe the console output to a file that you can search later:

   ```
   polyspace-bug-finder -sources aFile.c -compiler armclang -dump-preprocessing-info
   ```

   `aFile.c` can be a simple C file. You can also replace `polyspace-bug-finder` with `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server`.
2. Search for `__ARM_ARCH` in the file containing the console output. You can see the line with the macro definition:

   ```
   Remark: Definition of macro __ARM_ARCH (pre-processing __polyspace__stdstubs.c)
   |#define __ARM_ARCH 8
   |defined by syntax extension xml file
   |predefined macro
   ```

In this example, the macro is set to the value 8.

- To override this macro definition, use the option `Preprocessor definitions (-D)`.
- To undefine this macro, use the option `Disabled preprocessor definitions (-U)`.

## See Also

`Compiler (-compiler)`

### Topics

"Specify Polyspace Analysis Options"

# -generate-launching-script-for

Extract information from project file

## Syntax

`-generate-launching-script-for` *`PRJFILE`*

## Description

`-generate-launching-script-for` *`PRJFILE`* extracts information from a project file *`PRJFILE`* (created in the user interface of the Polyspace desktop products) so that you can run an analysis from the command line. For each project module and each configuration in each module, a folder is created containing the following files::

- `source_command.txt` — List of source files for the `-sources-list-file` option.
- `options_command.txt` — List of the analysis options for the `-options-file` option.
- `temporal_exclusions.txt` — List of temporal exclusions, generated only if you specify the `Temporally exclusive tasks (-temporal-exclusions-file)` option.
- `.polyspace_conf.psprj` — A copy of the project file Polyspace used to generate the scripting files.
- `launchingCommand.sh` (UNIX) or `launchingCommand.bat` (DOS) — shell script that calls the correct commands. The script also calls any options that cannot be given to the `-options-file` command, such as `-batch` or `-add-to-results-repository`. You can give this file additional analysis options as parameters.

After you set up a project in the Polyspace user interface, you can create this script from the resulting project file (with extension `.psprj`). The script that Polyspace generates runs the same analysis as a run in the user interface. If your project runs without errors in the Polyspace user interface, the script runs without errors at the command line.

To generate the script, you must run the command from the same folder as the project file.

## Examples

Extract information to run `myproject` from the command line. Use this option with the desktop binary `polyspace`:

- Bug Finder:

  ```
  polyspace -generate-launching-script-for myproject.psprj -bug-finder
  ```
- Code Prover:

  ```
  polyspace -generate-launching-script-for myproject.psprj
  ```

## See Also

### Topics

"Configure Polyspace Analysis Options in User Interface and Generate Scripts"

# -h | -help

Display list of possible options

## Syntax

```
-h
-help
```

## Description

`-h` and `-help` display the list of possible options in the command window along with option argument syntax.

## Examples

Display the command-line help:

- Bug Finder:

  ```
  polyspace-bug-finder -h
  polyspace-bug-finder -help
  ```
- Code Prover:

  ```
  polyspace-code-prover -h
  polyspace-code-prover -help
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -h
  polyspace-bug-finder-server -help
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -h
  polyspace-code-prover-server -help
  ```

## See Also

`-doc | -documentation`

# -I

Specify include folder for compilation

## Syntax

`-I folder`

## Description

`-I folder` specifies a folder that contains include files required for compiling your sources. You can specify only one folder for each instance of `-I`. However, you can specify this option multiple times.

The analysis looks for include files relative to the folder paths that you specify. For instance, if your code contains the preprocessor directive `#include<../mylib.h>` and you include the folder:

```
C:\My_Project\MySourceFiles\Includes
```

the folder `C:\My_Project\MySourceFiles` must contain a file `mylib.h`.

The analysis automatically includes the `./sources` folder (if it exists) after the include folders that you specify.

## Examples

Include two folders with the analysis:

- Bug Finder:

  ```
  polyspace-bug-finder -I /com1/inc -I /com1/sys/inc
  ```
- Code Prover:

  ```
  polyspace-code-prover -I /com1/inc -I /com1/sys/inc
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -I /com1/inc -I /com1/sys/inc
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -I /com1/inc -I /com1/sys/inc
  ```

The source folder is implicitly included. Include files in the source folder can be found automatically without explicit inclusion of the source folder with the `-I` option.

## Tips

- This option is useful for Polyspace analysis from the command line. In the Polyspace user interface, you add the include folders during project setup. See “Add Source Files for Analysis in Polyspace Desktop User Interface”.
- If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

- The option `-I` does not work recursively. If you add a folder path using this option, the analysis only considers include files that are directly in the folder *and not in subfolders of the folder*. If you also want subfolders in the include search path, explicitly add the subfolder paths using their own `-I`-s. For convenience, you can write a script that recursively looks through a folder and generates `-I`-s to subfolders of the folder.

## See Also

**Topics**
"Specify Polyspace Analysis Options"

# -ignore-code-annotations

Ignore C/C++ code annotations justifying Polyspace results and show all results as unreviewed

## Syntax

```
-ignore-code-annotations
```

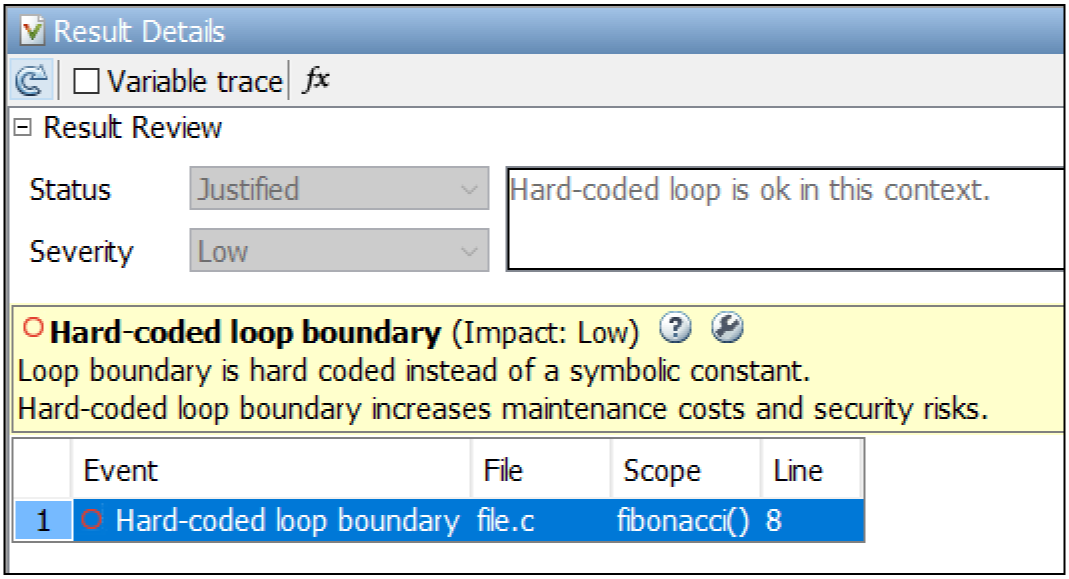
## Description

`-ignore-code-annotations` causes a Polyspace analysis to ignore code annotations justifying Polyspace results and show previously annotated results as unreviewed.

To avoid reviewing a result multiple times, you can add Polyspace-specific annotations to your code with review information such as justification for a result. Later runs take into account these annotations and prepopulate the annotated results with associated review information. However, in some cases, you might want to run a clean analysis as if the results have not been previously reviewed. You can use the option `-ignore-code-annotations` to run such an analysis with no history. The analysis ignores the code annotations and shows all annotated results without any review information taken from the annotations.

For instance, an analysis that takes into account code annotations shows these features:

- The **Results Details** pane for an annotated result is prepopulated with the review information. You can use filters to hide or show these annotated results.



Status: Justified

Severity: Low

Hard-coded loop is ok in this context.

○ **Hard-coded loop boundary** (Impact: Low)

Loop boundary is hard coded instead of a symbolic constant.

Hard-coded loop boundary increases maintenance costs and security risks.

| | Event | File | Scope | Line |
|---|---|---|---|---|
| 1 | ○ Hard-coded loop boundary | file.c | fibonacci() | 8 |

- Statistics on the **Dashboard** perspective in Polyspace Access use review information from the annotations. For instance, results that are justified using annotations are counted as **Done**.

If you use the option `-ignore-code-annotations`, both the individual results and the aggregated statistics do not include information from the code annotations.

For details on the code annotations workflow, see “Annotate Code and Hide Known or Acceptable Results”.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

To ignore previous code annotations, use the option `-ignore-code-annotations`.

- Bug Finder:

  ```
  polyspace-bug-finder -sources file.c -ignore-code-annotations
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -sources file.c -ignore-code-annotations
  ```
- Code Prover:

  ```
  polyspace-code-prover -sources file.c -ignore-code-annotations
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -sources file.c -ignore-code-annotations
  ```

## Tips

If you do not want to see any previous review information, you must also make sure that you are not importing review information from previous results. For instance, make sure that:

- You are not using options such as `-import-comments`.
- You have disabled automatic comment import from last results in the Polyspace user interface.

See also “Import Review Information from Previous Polyspace Analysis”.

# Version History

**Introduced in R2022a**

## See Also

### Topics

“Specify Polyspace Analysis Options”
“Annotate Code and Hide Known or Acceptable Results”

# -import-comments

Import review information from previous analysis

## Syntax

```
-import-comments resultsFolderPath
```

## Description

`-import-comments` *`resultsFolderPath`* imports the review information (status, severity and additional notes) from a previous analysis stored in the folder specified by the path *`resultsFolderPath`*.

You can import review information from the same type of results only. For instance:

- You cannot import review information from a results of a Bug Finder checker to a Code Prover run-time check. Even when the checker names sound similar, the underlying semantics of Bug Finder and Code Prover can be different. The only exception is checkers for coding rules. You can import comments between Bug Finder and Code Prover for coding rule violations.
- You cannot import review information from results of a file-by-file verification in Code Prover to results of a regular Code Prover verification.

You can also use this option to create a baseline for the analysis results. In the Polyspace user interface, if you click the **New** button, only the analysis results that are new compared to the baseline remain in the results list.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

Import review information from the previous results:

- Bug Finder:

  ```
  polyspace-bug-finder -sources filename -import-comments C:\Results\myProj\1.2
  ```
- Code Prover:

  ```
  polyspace-code-prover -sources filename -import-comments C:\Results\myProj\1.2
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -sources filename -import-comments C:\Results\myProj\1.2
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -sources filename -import-comments C:\Results\myProj\1.2
  ```

## Tips

If you use Polyspace as You Code extensions and you do not use previous analysis results that you downloaded from Polyspace Access, enter this option in an analysis options file. See options file.

## See Also

`-v[ersion]` | `polyspace-comments-import`

### Topics

“Import Review Information from Previous Polyspace Analysis”
“Baselining in Polyspace as You Code”

# -incremental

Enable incremental compilation during an analysis

## Syntax

```
-incremental
```

## Description

*This option affects only a Bug Finder analysis.*

`-incremental` enables an incremental mode during the compilation phase of a Polyspace analysis. Use this option to run faster multi-file analyses.

In this mode, after you run an initial analysis, Polyspace recompiles files during an analysis only if one of the following is true:

- You make edits to a source file. Polyspace recompiles all source files with edits since the last analysis.
- You make edits to a header file. Polyspace recompiles all files that declare a dependency on the edited header file.
- You make changes to the analysis options. Polyspace recompiles all files. Changes to these options are ignored:
  - `-author`
  - `-date`
  - `-prog`
  - `-verif-version`

When you use this option, make sure that you use the same results folder for all your analysis runs.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Example

Enable incremental mode for an analysis where you specify the list of sources in a file `sources.txt` and the list of analysis options in a file `opts.txt`.

- Polyspace Bug Finder

  ```
  polyspace-bug-finder -sources-list-file sources.txt -options-file opts.txt -incremental -results-dir .
  ```

- Polyspace Bug Finder Server

  ```
  polyspace-bug-finder-server -sources-list-file sources.txt -options-file opts.txt -incremental -results-dir .
  ```

## Tip

If you add a header to your project and a header with the same name already exists in a different folder, for instance you add `componentA/myHeader.h` when `componentB/myHeader.h` already exists, the new header does not trigger a recompilation.

# Version History

**Introduced in R2022a**

## See Also

**Topics**

"Specify Polyspace Analysis Options"

# -list-all-values

Display valid option arguments for a given command-line option

## Syntax

`-list-all-values` *`option`*

## Description

`-list-all-values` *`option`* displays all the valid option arguments for the command-line option *`option`*.

## Examples

Display the valid option arguments for option `-misra3`:

- Polyspace Bug Finder:

  ```
  polyspace-bug-finder -list-all-values -misra3
  ```
- Polyspace Code Prover:

  ```
  polyspace-code-prover -list-all-values -misra3
  ```
- Polyspace Bug Finder Server:

  ```
  polyspace-bug-finder-server -list-all-values -misra3
  ```
- Polyspace Code Prover Server:

  ```
  polyspace-code-prover-server -list-all-values -misra3
  ```

# Version History

**Introduced in R2020a**

## See Also

### Topics

“Specify Polyspace Analysis Options”

# -max-processes

Specify maximum number of processors for analysis

## Syntax

```
-max-processes num
```

## Description

`-max-processes` *`num`* specifies the maximum number of processes that you want the analysis to use. On a multicore system, the software parallelizes the analysis and creates the specified number of processes to speed up the analysis. The valid range of *`num`* is 1 to 128.

Unless you specify this option, a Code Prover verification uses up to four processes. If you have fewer than four processes, the verification uses the maximum available number. To increase or restrict the number of processes, use this option.

Unless you specify this option, a Bug Finder analysis uses the maximum number of available processes. Use this option to restrict the number of processes used.

To use this option effectively, determine the number of processors available for use. If the number of processes you create is greater than the number of processors available, the analysis does not benefit from the parallelization. Check the system information in your operating system.

Note that when you start a verification, a message states the number of logical processors detected on your system. However, the analysis is parallelized to the physical processor cores on a machine. Multithreading implementations such as hyper-threading is not taken into account.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

Disable parallel processing during the analysis:

- Bug Finder:

  ```
  polyspace-bug-finder -max-processes 1
  ```
- Code Prover:

  ```
  polyspace-code-prover -max-processes 1
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -max-processes 1
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -max-processes 1
  ```

## Tips

You must have at least 4 GB of RAM per process for analysis. For instance, if your machine has 16 GB of RAM, do not use this option to specify more than four processes.

This option is not useful in a Polyspace as You Code analysis.

## See Also

### Topics

"Specify Polyspace Analysis Options"

# -non-preemptable-tasks

Specify functions that represent nonpreemptable tasks

## Syntax

```
-non-preemptable-tasks function1[,function2[,...]]
```

## Description

`-non-preemptable-tasks` *`function1`*`[,`*`function2`*`[,...]]` specifies functions that represent nonpreemptable tasks.

The functions cannot be interrupted by other noncyclic tasks and cyclic tasks but can be interrupted by interrupts, preemptable or nonpreemptable. Noncyclic tasks are specified with the option `Tasks (-entry-points)`, cyclic tasks with the option `Cyclic tasks (-cyclic-tasks)` and interrupts with the option `Interrupts (-interrupts)`. For examples, see “Define Task Priorities for Data Race Detection in Polyspace”.

To specify a function as a nonpreemptable cyclic task, you must first specify the function as a cyclic or noncyclic task. The functions that you specify must have the prototype:

```
void function_name(void);
```

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

Consider the following program. Suppose that functions `task1` and `task2` represent tasks that can run concurrently.

```
int x;
void task1() {
    x++;
}
void task2() {
    x = 0;
}
void main() {
}
```

To specify that `task1` and `task2` are cyclic tasks, enter the following:

```
polyspace-bug-finder -sources filename -cyclic-tasks task1,task2
```

You see a **Data race** defect because the tasks attempt to concurrently access shared variable `x`.

Specify `task1` as nonpreemptable.

```
polyspace-bug-finder -sources multi.c -cyclic-tasks task1,task2
     -non-preemptable-tasks task1
```

The **Data race** disappears because the operation `x++` in `task1` cannot be interrupted anymore.

## Tips

- This option is not useful in a Polyspace as You Code analysis.
- Code Prover interprets this option with some limitations. The reason is that Code Prover considers all operations as potentially non-atomic and interruptible. This overapproximation leads to situations where the option might appear to be ignored. For an example, see "Effect of Task Priorities in Code Prover" (Polyspace Code Prover).

# Version History

**Introduced in R2016b**

## See Also

`Tasks (-entry-points)` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)` | `-preemptable-interrupts`

### Topics

"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Configuring Polyspace Multitasking Analysis Manually"
"Protections for Shared Variables in Multitasking Code"
"Define Task Priorities for Data Race Detection in Polyspace"
"Concurrency Defects"

# -options-file

Run Polyspace using list of options

## Syntax

`-options-file` *`file`*

## Description

`-options-file` *`file`* specifies the file that lists your analysis options. The file must be a text file with each option on a separate line. Use # to add comments to this file.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

When using this option in the user interface, in the **Other** field, specify the absolute path to your options file. For instance:

```
-options-file C:\psconfig\listofoptions.txt
```

When using this option in the command line, you can use relative path of the options file as the input.

## Examples

1 Create an options file called `listofoptions.txt` with your options. For example:

- Bug Finder or Bug Finder Server:

```
#These are the options for MyBugFinderProject
-lang c
-prog MyBugFinderProject
-author jsmith
-sources "mymain.c,funAlgebra.c,funGeometry.c"
-target x86_64
-compiler generic
-dos
-misra2 required-rules
-do-not-generate-results-for all-headers
-checkers default
-disable-checkers concurrency
-results-dir C:\Polyspace\MyBugFinderProject
```

- Code Prover or Code Prover Server:

```
#These are the options for MyCodeProverProject
-lang c
-prog MyCodeProverProject
-author jsmith
-sources "mymain.c,funAlgebra.c,funGeometry.c"
-target x86_64
-compiler generic
-dos
-misra2 required-rules
```

```
-do-not-generate-results-for all-headers
-main-generator
-results-dir C:\Polyspace\MyCodeProverProject
```

2 Run Polyspace using options in the file `listofoptions.txt`:

- Bug Finder:

  ```
  polyspace-bug-finder -options-file listofoptions.txt
  ```
- Code Prover:

  ```
  polyspace-code-prover -options-file listofoptions.txt
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -options-file listofoptions.txt
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -options-file listofoptions.txt
  ```

## See Also

**Topics**
"Specify Polyspace Analysis Options"

# -options-for-sources

Specify analysis options specific to a source file

## Syntax

`-options-for-sources` *`filename options`*

## Description

`-options-for-sources` *`filename options`* associates a semicolon-separated list of Polyspace analysis options with the source file specified by *`filename.`*.

This option is primarily used when the `polyspace-configure` command creates an options file for the subsequent Polyspace analysis. The option `-options-for-sources` associates a group of analysis options such as include folders and macro definitions with specific source files.

However, you can directly enter this option when manually writing options files. This option is useful in situations where you want to associate a group of options with a specific source file without applying it to other files.

In the user interface of the Polyspace desktop products, you can create a Polyspace project from your build command. The project uses the option `-options-for-sources` to associate specific Polyspace analysis options with specific files. However, when you open the project in the user interface, you cannot see the use of this option. Open the project in a text editor to see this option.

## Examples

In this sample options file, the include folder `/usr/lib/gcc/x86_64-linux-gnu/6/include` and the macros `__STDC_VERSION__` and `__GNUC__` are associated only with the source file `file.c` and not `fileAnother.c`.

```
-options-for-sources file.c;-I /usr/lib/gcc/x86_64-linux-gnu/6/include;
-options-for-sources file.c;-D __STDC_VERSION__=201112L;-D __GNUC__=6;
-sources file.c
-sources fileAnother.c
```

For the options used in this example, see:

- `-sources`
- `-I`
- `Preprocessor definitions (-D)`

## Tips

When associating multiple options with a source file, if you use an option separator other than semicolon, use a second option `-options-for-sources-delimiter` to explicitly specify this separator. For instance, if you use the separator `@`, specify the additional option:

```
-options-for-sources-delimiter @
```

Otherwise, the analysis assumes a semicolon separator.

## See Also

`-options-file` | `polyspace-configure`

### Topics

"Specify Polyspace Analysis Options"

# -preemptable-interrupts

Specify functions that represent preemptable interrupts

## Syntax

```
-preemptable-interrupts function1[,function2[,...]]
```

## Description

`-preemptable-interrupts` *`function1`*`[,`*`function2`*`[,...]]` specifies functions that represent preemptable interrupts.

The function acts as an interrupt in every way except that it can be interrupted by other interrupts, preemptable or nonpreemptable. Interrupts are specified with the option `Interrupts (-interrupts)`. For examples, see “Define Task Priorities for Data Race Detection in Polyspace”.

To specify a function as a preemptable interrupt, you must first specify the function as an interrupt. The functions that you specify must have the prototype:

```
void function_name(void);
```

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

Consider the following program. Suppose that functions `task1` and `task2` represent interrupts.

```
int x;
void task1() {
    x++;
}
void task2() {
    x = 0;
}
void main() {
}
```

To specify that `task1` and `task2` are cyclic tasks, enter the following:

```
polyspace-bug-finder -sources filename -interrupts task1,task2
```

You do not see a **Data race** defect because the operations in `task1` and `task2` cannot interrupt each other. Functions specified as interrupts are nonpreemptable by default.

Specify `task1` as preemptable.

```
polyspace-bug-finder -sources multi.c -interrupts task1,task2
     -preemptable-interrupts task1
```

A **Data race** now appears because the operation `x++` in `task1` can be interrupted by operations in `task2`.

## Tips

- This option is not useful in a Polyspace as You Code analysis.
- Code Prover interprets this option with some limitations. The reason is that Code Prover considers all operations as potentially non-atomic and interruptible. This overapproximation leads to situations where the option might appear to be ignored. For an example, see “Effect of Task Priorities in Code Prover” (Polyspace Code Prover).

# Version History

**Introduced in R2016b**

## See Also

`Tasks (-entry-points)` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)` | `-non-preemptable-tasks`

### Topics

“Specify Polyspace Analysis Options”
“Analyze Multitasking Programs in Polyspace”
“Configuring Polyspace Multitasking Analysis Manually”
“Protections for Shared Variables in Multitasking Code”
“Define Task Priorities for Data Race Detection in Polyspace”
“Concurrency Defects”

# -prog

Specify name of project

## Syntax

`-prog` *`projectName`*

## Description

`-prog` *`projectName`* specifies a name for your Polyspace project. This name must use only letters, numbers, underscores (_), dashes (-), or periods (.).

The name appears in the analysis log and a few other places.

## Examples

Assign a name to your Polyspace project:

- Bug Finder:

  ```
  polyspace-bug-finder -prog MyApp
  ```
- Code Prover:

  ```
  polyspace-code-prover -prog MyApp
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -prog MyApp
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -prog MyApp
  ```

## Tips

This option is not required for a Polyspace as You Code analysis.

## See Also

`-author` | `-date`

### Topics

“Specify Polyspace Analysis Options”

# -regex-replace-rgx -regex-replace-fmt

Make replacements in preprocessor directives

## Syntax

`-regex-replace-rgx` *`matchFileName`* `-regex-replace-fmt` *`replacementFileName`*

## Description

`-regex-replace-rgx` *`matchFileName`* `-regex-replace-fmt` *`replacementFileName`* replaces tokens in preprocessor directives for the purposes of Polyspace analysis. The original source code is unchanged. You match a token using a regular expression in the file *`matchFileName`* and replace the token using a replacement in the file *`replacementFileName`*.

Use the option only to replace or remove tokens in the preprocessor directives *before preprocessing*. Normally, if a token in your source code causes a compilation error, you can replace or remove the token from the preprocessed code by using the more convenient option `Command/script to apply to preprocessed files (-post-preprocessing-command)`. However, you cannot use the option to replace tokens in preprocessor directives. In this case, use `-regex-replace-rgx -regex-replace-fmt`.

For a complete list of regular expressions available with this option, see Perl documentation. Note that:

- Perl allows the syntax `s/`*`pattern`*`/`*`replacement`*`/`*`modifier`* for replacements. When using this option, you emulate this syntax only partially. You specify the pattern to match, *`pattern`*, in one file and its replacement, *`replacement`*, in another file. Search modifiers, that is, values of *`modifier`* in the Perl syntax, are not supported. For instance, by default, the option makes global replacements (that is, replaces the matched tokens wherever it finds them) and the matches are case-sensitive. You cannot change these defaults using search modifiers.
- The option supports both numbered and named capture groups. You can define a capture group in the match file by including it in parenthesis and use `$1`, `$2`, etc., to refer to those capture groups in the replacement file. Alternatively, you can also name the capture group and refer to the group by name. For an example, see "Replace Multiple Preprocessor Directives with Different Replacements Using Capture Groups" on page 3-57.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

In the user interface, specify absolute paths to the text files with the search and replace patterns.

## Examples

### Replace Undefined Symbols in Preprocessor Directive with Simpler Alternatives

Suppose that you want to modify this `#define` directive:

```
#define ROM_BEG_ADDR (uint16_t)(&_rom_beg)
```

to:

```
#define ROM_BEG_ADDR (0x4000u)
```

The reason for replacement might be that _rom_beg is undefined in the code provided to Polyspace and causes compilation issues. Since the tokens (uint16_t)(&_rom_beg) indicate an address and a Polyspace analysis does not keep track of precise addresses, you can replace (uint16_t)(&_rom_beg) with a simple address such as (0x4000u). To complicate the issue slightly, suppose also that you want to allow for one or more whitespace characters after #define and ROM_BEG_ADDR.

To make the replacement:

1. Specify this regular expression in a file match.txt:

   ```
   ^#define\s+ROM_BEG_ADDR\s+\(uint16_t\)\(\&_rom_beg\)
   ```

   These elements are used in the regular expression:

   - ^ asserts position at the start of a line.
   - \s+ represents one or more whitespace characters.

   The characters *, &, ( and ) are escaped with \.
2. Specify the replacement in a file replace.txt.

   ```
   #define ROM_BEG_ADDR \(0x4000u\)
   ```
3. Specify the two text files during analysis with the options -regex-replace-rgx and -regex-replace-fmt:

   - Bug Finder:

     ```
     polyspace-bug-finder -sources filename
                                 -regex-replace-rgx match.txt
                                 -regex-replace-fmt replace.txt
     ```
   - Code Prover:

     ```
     polyspace-code-prover -sources filename
                                 -regex-replace-rgx match.txt
                                 -regex-replace-fmt replace.txt
     ```
   - Bug Finder Server:

     ```
     polyspace-bug-finder-server -sources filename
                                 -regex-replace-rgx match.txt
                                 -regex-replace-fmt replace.txt
     ```
   - Code Prover Server:

     ```
     polyspace-code-prover-server -sources filename
                                 -regex-replace-rgx match.txt
                                 -regex-replace-fmt replace.txt
     ```

### Replace Multiple Preprocessor Directives with Different Replacements Using Capture Groups

The following example defines two macros bypass_UInt16_ and bypass_UInt32_, both of which contains undefined symbols, UInt16_DO_NOT_EXIST and UInt32_DO_NOT_EXIST.

```
typedef unsigned short UInt16;
typedef signed short Int16;
```

```
typedef unsigned int UInt32;
typedef signed int Int32;

UInt16 x16;
Int16 y16, z16;

UInt32 x32;
Int32 y32, z32;

#define bypass_UInt16_(_var, _value, _add) _var = _value +/*CTO*/(UInt16_DOES_NOT_EXIST) _add
#define bypass_UInt32_(_var, _value, _add) _var = _value +/*CTO*/(UInt32_DOES_NOT_EXIST) _add

void main(void){
    bypass_UInt16_(x16, y16, z16);
    bypass_UInt32_(x32, y32, z32);
}
```

Both undefined symbols follow a comment `/* CTO */`. Since the symbols are undefined, they cause compilation errors. Suppose that you want to modify the macro definitions so that the undefined symbols appear inside the comments instead of just following it. Since you want to perform similar modifications to both definitions but do not want to replace both undefined symbols with one replacement, you can use capture groups to keep the symbol names intact.

To make the replacements:

1 Specify this regular expression in a file `match.txt`:

   ```
   \/\* CTO \*\/(\s+\(.*?\))
   ```

   These elements are used in the regular expression:

   - The characters `/`, `*`, ( and ) are escaped with `\`. For instance, to match against:

     ```
     /* CTO */
     ```

     You enter the escaped sequence:

     ```
     \/\* CTO \*\/
     ```

   - The outer parenthesis in the sequence:

     ```
     (\s+\(.*?\))
     ```

     indicates a capture group. The group includes one or more whitespace characters (indicated with `\s+`) and one or more characters inside a parenthesis. This capture group can capture both the undefined symbols, `(UInt16_DO_NOT_EXIST)` and `(UInt32_DO_NOT_EXIST)`.

     This capture group needs to be referred by number in the replacement file. Alternatively, you can create a named capture group and refer to the group by name in the replacement file. For instance, the following regular expression creates a named capture group `cto_group`:

     ```
     \/\* CTO \*\/(?<cto_group>\s+\(.*?\))
     ```

2 Specify the replacement in a file `replace.txt`.

   ```
   /* CTO $1 */
   ```

   The `$1` refers to the previously captured group. This replacement simply places the captured groups before the `*/` closing the comments.

   If you created a capture group named `cto_group` in the match file, enter this replacement:

   ```
   /* CTO $+{cto_group} */
   ```

The name of the capture group is included in `$+{ }`.

3. Specify the two text files during analysis with the options `-regex-replace-rgx` and `-regex-replace-fmt` (as shown in previous example).

You see the following code in the analysis results with the undefined symbols now included in comments:

```
typedef unsigned short UInt16;
typedef signed short Int16;

typedef unsigned int UInt32;
typedef signed int Int32;

UInt16 x16;
Int16 y16, z16;

UInt32 x32;
Int32 y32, z32;

#define bypass_UInt16_(_var, _value, _add) _var = _value + /* CTO  (UInt16_DO_NOT_EXIST) */ _add
#define bypass_UInt32_(_var, _value, _add) _var = _value + /* CTO  (UInt32_DO_NOT_EXIST) */ _add

void main(void){
    bypass_UInt16_(x16, y16, z16);
    bypass_UInt32_(x32, y32, z32);
}
```

## Tips

- If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.
- To make replacements in multiple kinds of preprocessor directives, enter one regular expression per line in the match file and its replacement on the corresponding line in the replacement file. Each preprocessor line that matches a regular expression in the match file is replaced with the corresponding replacement from the replacement file. You can also enter all the matches and replacements in one line separated by `|`. However, one entry per line improves the readability of the files.

  For instance, the match file can contain two regular expressions such as:

  ```
  ^#define\s+ROM_BEG_ADDR\s+\(uint16_t\)\(\&_rom_beg\)
  ^#define\s+ROM_END_ADDR\s+\(uint16_t\)\(\&_rom_end\)
  ```

  And the replacement file can contain these two replacements:

  ```
  #define ROM_BEG_ADDR \(0x4000u\)
  #define ROM_END_ADDR \(0x8000u\)
  ```

  With these matches and replacements, the following source code:

  ```
  #include <stdint.h>

  #define ROM_BEG_ADDR (uint16_t)(&_rom_beg)
  #define ROM_END_ADDR (uint16_t)(&_rom_end)
  ```

```
void main() {
    uint16_t beg_addr = ROM_BEG_ADDR;
    uint16_t end_addr = ROM_END_ADDR;
}
```

is converted to the following preprocessed code before analysis:

```
#include <stdint.h>

#define ROM_BEG_ADDR (0x4000u)
#define ROM_END_ADDR (0x8000u)


void main() {
    uint16_t beg_addr = ROM_BEG_ADDR;
    uint16_t end_addr = ROM_END_ADDR;
}
```

## See Also

`Command/script to apply to preprocessed files (-post-preprocessing-command)`

### Topics

"Specify Polyspace Analysis Options"

# -report-output-name

Specify name of report

## Syntax

`-report-output-name` *`reportName`*

## Description

`-report-output-name` *`reportName`* specifies the name of an analysis report.

The default name for a report is *`Prog_Template.Format`*:

- *`Prog`* is the name of the project specified by `-prog`.
- *`TemplateName`* is the type of report template specified by `-report-template`.
- *`Format`* is the file extension for the report specified by `-report-output-format`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

Specify the name of the analysis report:

- Bug Finder:

  ```
  polyspace-bug-finder -report-template Developer
        -report-output-name Airbag_v3.doc
  ```
- Code Prover:

  ```
  polyspace-code-prover -report-template Developer
        -report-output-name Airbag_v3.doc
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -report-template Developer
        -report-output-name Airbag_v3.doc
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -report-template Developer
        -report-output-name Airbag_v3.doc
  ```

## Tips

You cannot generate reports with Polyspace as You Code.

## See Also

`Bug Finder and Code Prover report (-report-template)` | `Output format (-report-output-format)`

**Topics**

“Specify Polyspace Analysis Options”

“Generate Reports from Polyspace Results”

# -results-dir

Specify the results folder

## Syntax

```
-results-dir resultsFolder
```

## Description

`-results-dir resultsFolder` specifies where to save the analysis results. The default location at the command line is the current folder.

Note that the results folder is cleaned up and repopulated at each run. To avoid accidental removal of files during the cleanup, instead of using an existing folder that contains other files, specify a dedicated folder for the Polyspace results.

If you are running analysis in the user interface of the Polyspace desktop products, see “Run Analysis in Polyspace Desktop User Interface”.

## Examples

Specify to store your results in the `RESULTS` folder:

- Bug Finder:

  ```
  polyspace-bug-finder -results-dir RESULTS
  ```
- Code Prover:

  ```
  polyspace-code-prover -results-dir RESULTS
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -results-dir RESULTS
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -results-dir RESULTS
  ```

You can create the name of the results folder based on the verification date and time. For instance, in a Bash shell, enter these commands to create a variable `RESULTS` that begins with `results_` and contains the current date and time:

```
export DATETIME=$(date +%d%B_%HH%M_%A)
export RESULTS=results_$DATE
```

You can then use the variable `RESULTS` as argument of the option `-results-dir`:

```
-results-dir $RESULTS
```

## Tips

If you use Polyspace as You Code extensions in IDEs, this option is implemented through the IDE extension setting. You do not have to use this option explicitly. If you want to explicitly use this option, enter the option in an analysis options file. See options file.

## See Also

### Topics

“Specify Polyspace Analysis Options”

# -scheduler

Specify cluster or job scheduler

## Syntax

`-scheduler` *`schedulingOption`*

## Description

`-scheduler` *`schedulingOption`* specifies the head node of the MATLAB Parallel Server cluster that manages Polyspace analysis submissions from multiple clients and allocates the analysis to worker nodes. You use this option along with the option `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)` to offload an analysis from a desktop to a remote cluster. Note that you use this option with the commands in the desktop products (`polyspace-bug-finder` and `polyspace-code-prover`) and not the commands in the server products (`polyspace-bug-finder-server` and `polyspace-code-prover-server`).

For more information, see "Install Products for Submitting Polyspace Analysis from Desktops to Remote Server".

## Examples

Run a batch analysis on a remote server using one of these syntaxes for the job scheduler:

- Bug Finder:

  ```
  polyspace-bug-finder -batch -scheduler NodeHost
  polyspace-bug-finder -batch -scheduler 192.168.1.124:12400
  polyspace-bug-finder -batch -scheduler MJSName@NodeHost
  ```
- Code Prover:

  ```
  polyspace-code-prover -batch -scheduler NodeHost
  polyspace-code-prover -batch -scheduler 192.168.1.124:12400
  polyspace-code-prover -batch -scheduler MJSName@NodeHost
  ```

For details, see "Send Polyspace Analysis from Desktop to Remote Servers Using Scripts".

You can track the status of the job using the `polyspace-jobs-manager` command:

```
polyspace-jobs-manager listjobs -scheduler NodeHost
```

## Tips

You cannot submit analysis jobs to a remote cluster with Polyspace as You Code.

## See Also

`Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`

### Topics

"Send Polyspace Analysis from Desktop to Remote Servers Using Scripts"

“Send Bug Finder Analysis from Desktop to Locally Hosted Server”
“Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”

# -sources

Specify source files

## Syntax

```
-sources file1[,file2,...]
-sources file1 -sources file2
```

## Description

`-sources file1[,file2,...]` or `-sources file1 -sources file2` specifies the list of source files that you want to analyze. You can use standard UNIX wildcards with this option to specify your sources.

The source files are compiled in the order in which they are specified.

## Examples

Analyze the files `mymain.c`, `funAlgebra.c`, and `funGeometry.c`.

- Bug Finder:

  ```
  polyspace-bug-finder -sources mymain.c
       -sources funAlgebra.c -sources funGeometry.c
  ```
- Code Prover:

  ```
  polyspace-code-prover -sources mymain.c
       -sources funAlgebra.c -sources funGeometry.c
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -sources mymain.c
       -sources funAlgebra.c -sources funGeometry.c
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -sources mymain.c
       -sources funAlgebra.c -sources funGeometry.c
  ```
- Polyspace as You Code:

  ```
  polyspace-bug-finder-access -sources myfile.c
  ```

  Note that you can only analyze one file at a time with Polyspace as You Code. If you use Polyspace as You Code extensions in IDEs, you do not have to specify this option. The analysis runs on the file that is currently active in your IDE.

## Tips

This option is useful for Polyspace analysis from the command line. In the Polyspace user interface, you add the source files during project setup. See “Add Source Files for Analysis in Polyspace Desktop User Interface”.

## See Also

`-sources-list-file` | `polyspace-configure`

### Topics

"Specify Polyspace Analysis Options"

# -sources-list-file

Specify file containing list of sources

## Syntax

`-sources-list-file *file_path*`

## Description

`-sources-list-file` *`file_path`* specifies the absolute path to a text file that lists each file name that you want to analyze.

To specify your sources in the text file, on each line, specify the path to a source file. You can specify an absolute path or a path relative to the folder from which you are running the analysis. For example:

```
C:\Sources\myfile.c
C:\Sources2\myfile2.c
```

## Examples

Run analysis on files listed in `files.txt`:

- Bug Finder:

  ```
  polyspace-bug-finder -sources-list-file "C:\Analysis\files.txt"
  polyspace-bug-finder -sources-list-file "/home/polyspace/files.txt"
  ```
- Code Prover:

  ```
  polyspace-code-prover -sources-list-file "C:\Analysis\files.txt
  polyspace-code-prover -sources-list-file "/home/polyspace/files.txt"
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -sources-list-file "C:\Analysis\files.txt"
  polyspace-bug-finder-server -sources-list-file "/home/polyspace/files.txt"
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -sources-list-file "C:\Analysis\files.txt
  polyspace-code-prover-server -sources-list-file "/home/polyspace/files.txt"
  ```

## Tips

You cannot use this option with Polyspace as You Code.

## See Also

### Topics

“Specify Polyspace Analysis Options”

# -submit-job-from-previous-compilation-results

Specify that the analysis job must be resubmitted without recompilation

## Syntax

```
-submit-job-from-previous-compilation-results
```

## Description

`-submit-job-from-previous-compilation-results` specifies that the Polyspace analysis must start after the compilation phase with compilation results from a previous analysis. The option is primarily useful when offloading a Polyspace analysis from desktops to remote servers. If a remote analysis stops after compilation, for instance because of communication problems between the server and client computers, use this option. Note that you use this option with the commands in the desktop products (`polyspace-bug-finder` and `polyspace-code-prover`) and not the commands in the server products (`polyspace-bug-finder-server` and `polyspace-code-prover-server`).

When you perform a remote analysis:

1. On the local host computer, the Polyspace software performs code compilation and coding rule checking.
2. The analysis job is then submitted to the MATLAB job scheduler on the head node of the MATLAB Parallel Server cluster.
3. The head node of the MATLAB Parallel Server cluster assigns the verification job to a worker node, where the remaining phases of the Polyspace analysis occur.

If an analysis stops after completing the first step and you restart the analysis, use this option to reuse compilation results from the previous analysis. You thereby avoid restarting the analysis from the compilation phase.

If previous compilation results do not exist in the current folder, an error occurs. Remove the option and restart analysis from the compilation phase.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

Specify remote analysis with compilation results from a previous analysis:

- Bug Finder:

```
polyspace-bug-finder -batch -scheduler localhost
       -submit-job-from-previous-compilation-results
```

- Code Prover:

```
polyspace-code-prover -batch -scheduler localhost
       -submit-job-from-previous-compilation-results
```

## Tips

You cannot submit analysis jobs to a remote cluster with Polyspace as You Code.

## See Also

**Topics**
“Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”
“Send Bug Finder Analysis from Desktop to Locally Hosted Server”
“Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”

# -termination-functions

Specify process termination functions

## Syntax

`-termination-functions` *`function1`*`[,`*`function2`*`[,...]]`

## Description

`-termination-functions` *`function1`*`[,`*`function2`*`[,...]]` specifies functions that behave like the exit function and terminate your program.

Use this option to specify program termination functions that are declared but not defined in your code.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

Polyspace detects an **Integer division by zero** defect in the following code because it does not recognize that `my_exit` terminates the program.

```
void my_exit(void);

void main() {
    double ans;
    ans = reciprocal(1);
    ans = reciprocal(0);
}

double reciprocal(int val) {
  if(val==0)
    my_exit();
  return (1/val);
}
```

To prevent Polyspace from flagging the division operation, use the `-termination-functions` option:

```
polyspace-bug-finder -termination-functions my_exit
```

## See Also

`polyspaceBugFinder`

### Topics

“Run Polyspace Analysis from Command Line”

# -tmp-dir-in-results-dir

Keep temporary files in results folder

## Syntax

```
-tmp-dir-in-results-dir
```

## Description

`-tmp-dir-in-results-dir` specifies that temporary files must be stored in a subfolder of the results folder. Use this option only when the standard temporary folder does not have enough disk space. If the results folder is mounted on a network drive, this option can slow down your processor.

To learn how Polyspace determines the temporary folder location, see “Storage of Temporary Files During Polyspace Analysis”.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

## Examples

Store temporary files in the results folder:

- Bug Finder:

  ```
  polyspace-bug-finder -tmp-dir-in-results-dir
  ```
- Code Prover:

  ```
  polyspace-code-prover -tmp-dir-in-results-dir
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -tmp-dir-in-results-dir
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -tmp-dir-in-results-dir
  ```

## Tips

If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

## See Also

### Topics

“Specify Polyspace Analysis Options”

# -v | -version

Display Polyspace version number

## Syntax

```
-v
-version
```

## Description

`-v` or `-version` displays the version number of your Polyspace product.

## Examples

Display the version number and release of your Polyspace product:

- Bug Finder:

  ```
  polyspace-bug-finder -v
  ```
- Code Prover:

  ```
  polyspace-code-prover -v
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -v
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -v
  ```

# -verif-version

Assign a version identifier

## Syntax

`-verif-version` *`id`*

## Description

`-verif-version` *`id`* assigns an identifier, *`id`*, to identify the analysis. You can use this identifier to refer to different analyses at the command line. For example, you can import comments from a previous analysis using the identifier.

## Examples

Assign a verification identifier:

- Bug Finder:

  ```
  polyspace-bug-finder -verif-version 1.3
  ```
- Code Prover:

  ```
  polyspace-code-prover -verif-version 1.3
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -verif-version 1.3
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -verif-version 1.3
  ```

## Tips

This option is not useful for Polyspace as You Code.

## See Also

**Topics**
"Specify Polyspace Analysis Options"

# -xml-annotations-description

Apply custom code annotations to Polyspace analysis results

## Syntax

`-xml-annotations-description` *`file_path`*

## Description

`-xml-annotations-description` *`file_path`* uses the annotation syntax defined in the XML file located in *`file_path`* to interpret code annotations in your source files. You can use the XML file to specify an annotation syntax and map it to the Polyspace annotation syntax. When you run an analysis by using this option, you can justify and hide results with annotations that use your syntax. If you run Polyspace at the command line, *`file_path`* is the absolute path or path relative to the folder from which you run the command. If you run Polyspace through the user interface, *`file_path`* is the absolute path.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

If you use Polyspace as You Code extensions in IDEs, enter this option in an analysis options file. See options file.

### Why Use This Option

If you have existing annotations from previous code reviews, you can import these annotations to Polyspace. You do not have to review and justify results that you have already annotated. Similarly, if your code comments must adhere to a specific format, you can map and import that format to Polyspace.

## Examples

### Import Existing Annotations for Coding Rule Violations

Suppose that you have previously reviewed source file `zero_div.c` containing the following code, and justified certain MISRA C: 2012 violations by using custom annotations.

```
#include <stdio.h>

/* Violation of Misra C:2012
rules 8.4 and 8.7 on the next
line of code. */

int func(int p) //My_rule 50, 51
{
    int i;
    int j = 1;

    i = 1024 / (j - p);
    return i;
}

/* Violation of Misra C:2012
rule 8.4 on the next line of
code */

int func2(void){ //My_rule 50
    int x=func(2);
    return x;
}
```

The code comments **`My_rule 50, 51`** and **`My_rule 50`** do not use the Polyspace annotation syntax. Instead, you use a convention where you place all MISRA rules in a single numbered list. In this list, rules 8.4 and 8.7 correspond to the numbers 50 and 51.You can check this code for MISRA C: 2012 violations by typing the command:

- Bug Finder:

  `polyspace-bug-finder -sources *source_path* -misra3 all`
- Code Prover:

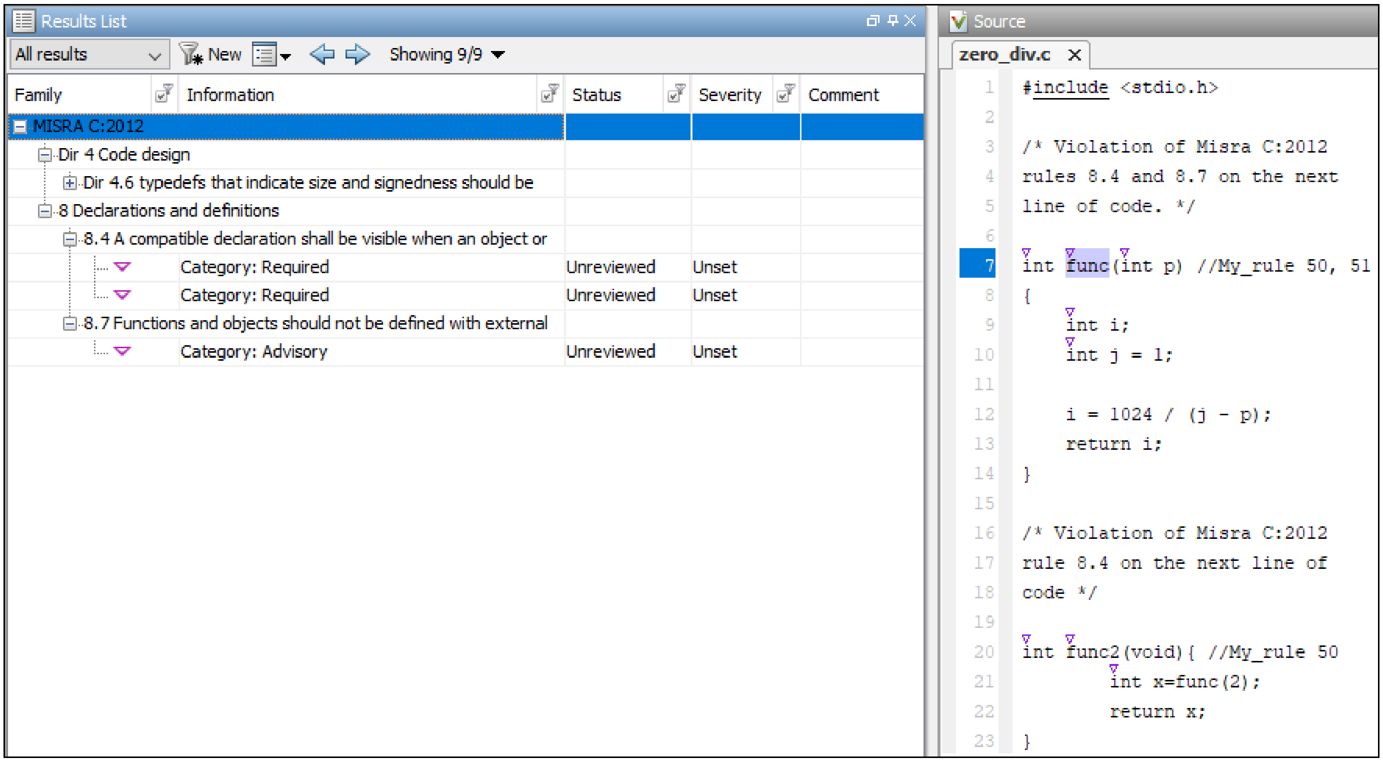
  `polyspace-code-prover -sources *source_path* -misra3 all -main-generator`
- Bug Finder Server:

  `polyspace-bug-finder-server -sources *source_path* -misra3 all`
- Code Prover Server:

  `polyspace-code-prover-server -sources *source_path* -misra3 all -main-generator`

*`source_path`* is the path to `zero_div.c`.

The annotated violations appear in the **Results List** pane. You must review and justify them again.



| Family | Information | Status | Severity | Comment |
|---|---|---|---|---|
| MISRA C:2012 | | | | |
| Dir 4 Code design | | | | |
| Dir 4.6 typedefs that indicate size and signedness should be | | | | |
| 8 Declarations and definitions | | | | |
| 8.4 A compatible declaration shall be visible when an object or | | | | |
| | Category: Required | Unreviewed | Unset | |
| | Category: Required | Unreviewed | Unset | |
| 8.7 Functions and objects should not be defined with external | | | | |
| | Category: Advisory | Unreviewed | Unset | |

```
#include <stdio.h>

/* Violation of Misra C:2012
rules 8.4 and 8.7 on the next
line of code. */

int func(int p) //My_rule 50, 51
{
    int i;
    int j = 1;

    i = 1024 / (j - p);
    return i;
}

/* Violation of Misra C:2012
rule 8.4 on the next line of
code */

int func2(void){ //My_rule 50
        int x=func(2);
        return x;
}
```

This XML example defines the annotation format used in `zero_div.c` and maps it to the Polyspace annotation syntax:

- The format of the annotation is the keyword `My_rule`, followed by a space and one or more comma-separated alphanumeric rule identifiers.
- Rule identifiers 50 and 51 are mapped to MISRA C: 2012 rules 8.4 and 8.7 respectively. The mapping uses the Polyspace annotation syntax.

```
<?xml version="1.0" encoding="UTF-8"?>

<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
             Group="exampleCustomAnnotation">

  <Expressions Search_For_Keywords="My_rule"
              Separator_Result_Name="," >


    <!-- This section defines the annotation syntax format -->
   <Expression Mode="SAME_LINE"
               Regex="My_rule\s(\w+(\s*,\s*\w+)*)"
               Rule_Identifier_Position="1"
               />

 </Expressions>
  <!-- This section maps the user annotation to the Polyspace
  annotation syntax -->
 <Mapping>
 <Result_Name_Mapping  Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
 <Result_Name_Mapping  Rule_Identifier="51" Family="MISRA-C3" Result_Name="8.7"/>
 </Mapping>
</Annotations>
```

To import the existing annotations and apply them to the corresponding Polyspace results:

1 Copy the preceding code example to a text editor and save it on your machine as `annotations_description.xml`, for instance in `C:\Polyspace_workspace\annotations\`.

2 Rerun the analysis on `zero_div.c` by using the command:

- Bug Finder:

```
polyspace-bug-finder -sources source_path -misra3 all ^
-xml-annotations-desription ^
C:\Polyspace_workspace\annotations\annotations_description.xml
```

- Code Prover:

```
polyspace-code-prover -sources source_path -misra3 all ^
-main-generator -xml-annotations-desription ^
C:\Polyspace_workspace\annotations\annotations_description.xml
```

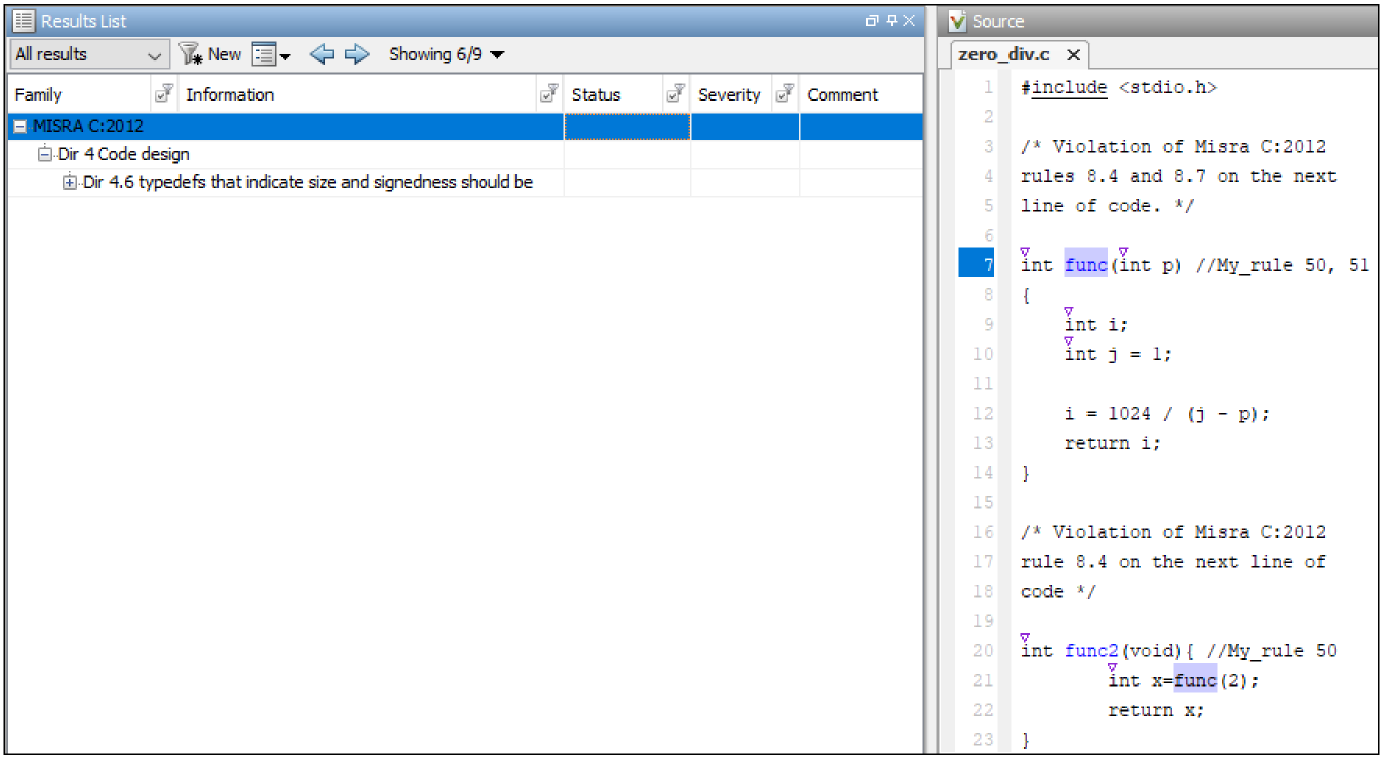
- Bug Finder Server:

```
polyspace-bug-finder-server -sources source_path -misra3 all ^
-xml-annotations-desription ^
C:\Polyspace_workspace\annotations\annotations_description.xml
```

- Code Prover Server:

```
polyspace-code-prover-server -sources source_path -misra3 all ^
-main-generator -xml-annotations-desription ^
C:\Polyspace_workspace\annotations\annotations_description.xml
```

Polyspace considers the annotated results justified and hides them in the **Results List** pane.



| Family | Information | Status | Severity | Comment |
|---|---|---|---|---|
| MISRA C:2012 | | | | |
| Dir 4 Code design | | | | |
| Dir 4.6 typedefs that indicate size and signedness should be | | | | |

```
#include <stdio.h>

/* Violation of Misra C:2012
rules 8.4 and 8.7 on the next
line of code. */

int func(int p) //My_rule 50, 51
{
    int i;
    int j = 1;

    i = 1024 / (j - p);
    return i;
}

/* Violation of Misra C:2012
rule 8.4 on the next line of
code */

int func2(void){ //My_rule 50
        int x=func(2);
        return x;
}
```

## Version History

**Introduced in R2017b**

## See Also

### Topics

"Specify Polyspace Analysis Options"
"Define Custom Annotation Format"
"Annotation Description Full XML Template"
"Fix Errors Applying Custom Annotation Format for Polyspace Results"

# Polyspace DOS/UNIX Commands

# Polyspace DOS/Unix Commands

# admin-docker-agent

(DOS/UNIX) Launch Cluster Admin interface to manage User Manager, Issue Tracker, and Polyspace Access Apps

## Syntax

```
admin-docker-agent [OPTIONS]
```

## Description

`admin-docker-agent [OPTIONS]` starts the **Cluster Admin** interface. If you do not specify additional `OPTIONS`, the Admin agent uses host name `localhost` and starts with the `HTTP` protocol on port `9443`.

## Input Arguments

### `OPTIONS` — Options to manage the Cluster Admin

string

Options to specify and manage the connection settings of the **Cluster Admin**.

**General Options**

| Option | Description |
|---|---|
| `--hostname` *`hostName`* | Specify the fully qualified domain name of the machine on which you run the **Cluster Admin**. This option is required if you use the HTTPS configuration options. *`hostName`* must match the `Common Name (CN)` that you specify to obtain SSL certificates.<br>The default host name is `localhost`. |
| `--port` *`portNumber`* | Specify the server port number that you use to access the **Cluster Admin** web interface.<br>The default port value is 9443. |
| `--data-dir` *`dirPath`* | Specify the absolute path to the folder containing the `settings.json` file.<br>If the file does not exist, the **Cluster Admin** creates it in the specified folder.<br>If the file already exists, the **Cluster Admin** reuses its contents to configure the settings.<br>The default folder is the current folder. |

| Option | Description |
|---|---|
| `--network-name` *`networkName`* | Specify the name of the Docker network that the Polyspace Access, User Manager, and Issue Tracker apps use.<br>Use this option if you do not want the apps to use the default `mathworks` network, for instance, if that network conflicts with an existing network.<br>To check if your docker network conflicts with an existing network, run this command:<br>`docker network inspect` *`networkName`*<br>and inspect the `IPAM.Config` node to view the IP range that is used by the network. You might need to contact your network administrator to determine if the IP range is used by other services. To find *`networkName`*, use the command `docker network ls`.<br>To create a new network, see docker network create. |
| `--force-exposing-ports` | Specify this option to expose the ports of the services when you install all the services on a single node. To specify the Docker host port to which the exposed ports bind, open the **Cluster Admin**, click **Configure Nodes**, then go to the **Services** tab.<br>By default, when you install on a single node, the ports of the services are not exposed .<br>Use this option if you install on a single node but you must communicate with one of the services through a third party tool, for instance if you use PostgreSQL utilities to communicate with the Polyspace Access database. |
| `--reset-password` | Reset the password that you use to log into the **Cluster Admin** web interface. |
| `--version` | Display the version number of the Admin agent. |
| `--help` | Display the help menu. |

**HTTPS Configuration Options**

On Windows systems, all paths must point to local drives.

| Option | Description |
|---|---|
| `--ssl-cert-file` *`absolutePath`* | Specify the absolute path to the SSL certificate `PEM` file. |
| `--ssl-key-file` *`absolutePath`* | Specify the absolute path to the SSL private key `PEM` file that you used to generate the certificate. |

| Option | Description |
|---|---|
| `--ssl-ca-file` *`absolutePath`* | Specify the absolute path to the certificate store where you store trusted certificate authorities. For instance, on a Linux Debian® distribution, `/etc/ssl/certs/ca-certificates.crt` .<br>If you use self-signed certificates, use the same file that you specify for `--ssl-cert-file` |
| `--restart-gateway` | Use this option to restart the **Gateway** service if you restart the `admin-docker-agent` and you make changes to the HTTPS configuration options or you specify a different port.<br>Restart the **Gateway** service by using this option if you make changes to the HTTPS configuration options or specify a different port. |

**New Node Configuration Options**

If you choose to install Polyspace Access on multiple machines, use these options to create nodes on the different machines. In the **Cluster Dashboard**, click **Configure Nodes**, and then select the **Services** tab to select the node on which you want to run the service.

Before you create a node, you must have an instance of the `admin-docker-agent` already running on at least one other machine. This other machine hosts the `primary` node.

| Option | Description |
|---|---|
| `--master-host` *`hostName:port`* | Specify the host name and port number of the machine hosting the `primary` node. |
| `--node-id` *`nodeName`* | Name of the node that you create. After you start the **Cluster Admin**, you see this node listed in the **Node** drop-down lists on the **Services** tab of the **Nodes** settings. |

## Examples

### Configure HTTPS Protocol With Self-Signed Certificate

The **Cluster Admin** uses the HTTP protocol by default. Encrypt the data between the **Cluster Admin** and client machines by configuring the **Cluster Admin** with the HTTPS protocol. This configuration also enables HTTPS for the API Gateway service, which handles communications between all the other services and client machines.

If you install Polyspace Access on multiple nodes, or if you use the `--force-exposing-ports` to start the **Admin** agent, you must configure HTTPS for the User Manager, Issue Tracker, and Polyspace Access services separately. To configure HTTPS for the services, click **Configure Nodes** on the **Cluster Dashboard**.

Create a self-signed SSL certificate and private key file by using the `openssl` toolkit.

```
openssl req -newkey rsa:2048 -new -nodes -x509 -days 365 -keyout self_key.pem -out self_cert.pem
```

After you enter the command, follow the prompts on the screen. You can leave most fields blank, but you must provide a `Common Name` (CN). The CN must match the fully qualified domain name (FQDN) of the machine running the `admin-docker-agent`. The command outputs a certificate file `self_cert.pem` and a private key file `self_key.pem`.

To obtain the FQDN of the machine, use the command `hostname --fqdn` on Linux or `net config workstation | findstr /C:"Full Computer name"` on Windows .

Start the `admin-docker-agent` by using the certificate and private key files that you generated and specify *hostName*, the FQDN of the machine. *hostName* must match the FQDN that you entered for the CN of the SSL certificate. In the command, specify the absolute path to the files.

| | |
|---|---|
| **Windows PowerShell** | admin-docker-agent --hostname *hostName*`<br>--ssl-cert-file *absolutePathTo*\self_cert.pem `<br>--ssl-key-file *absolutePathTo*\self_key.pem.pem `<br>--ssl-ca-file *absolutePathTo*\self_cert.pem |
| **Linux** | ./admin-docker-agent --hostname *hostName* \\<br>--ssl-cert-file *absolutePathTo*/self_cert.pem \\<br>--ssl-key-file *absolutePathTo*/self_key.pem.pem \\<br>--ssl-ca-file *absolutePathTo*/self_cert.pem |

You can now access the **Cluster Admin** web interface from your browser by using https://*hostName*:9443/admin.

# Version History

**Introduced in R2020b**

## See Also

### Topics

"Configure and Start the Cluster Admin"

# polyspace-access

(DOS/UNIX) Manage Polyspace Access results and users at the command line

## Syntax

```
polyspace-access -create-project folderPath -host hostname [login options]
polyspace-access -move-project sourcePath -to-project-path destinationPath -
host hostname [login options]
polyspace-access -delete-project projectToDelete -host hostname [login
options]
polyspace-access -list-project [projectPath] -host hostname [login options]
polyspace-access -list-runs projectPath -host hostname [login options]

polyspace-access -upload pathToFileOrFolderOrZipFile [upload options] -host
hostname [login options]
polyspace-access -export findingsToExport -output filePath [export options] -
host hostname [login options]
polyspace-access -download findingsToDownload -output-folder-path
outputFolderPath -host hostname [login options]

polyspace-access -review fileOrFindingID -project-path projectToReview review
options -host hostname [login options]
polyspace-access -import-reviews sourceProjectPath -project-path
targetProjectPath [-import-strategy never-overwrite-target | always-
overwrite-target] -host hostname [login options]

polyspace-access -set-sqo projectPathWithSQO -level sqoLevel [-name sqoName]
-host hostname [login options]
polyspace-access -get-sqo projectPathWithSQO -host hostname [login options]
polyspace-access -list-sqo -host hostname [login options]

polyspace-access -add-label label -run-id runID -host hostname [login
options]
polyspace-access -remove-label label -run-id runID -host hostname [login
options]
polyspace-access -set-unassigned-findings findingsToAssign -owner
userToAssign -source-contains pattern [set unassigned findings options] -host
hostname [login options]

polyspace-access -set-role role -user username -group groupname -project-path
folderPathOrProjectPath -host hostname [login options]
polyspace-access -unset-role -user username -group groupname -project-path
folderPathOrProjectPath -host hostname [login options]

polyspace-access -generate-migration-commands metrics_dir -output-folder-path
dir [generate migration commands options]
polyspace-access -migrate -option-file-path dir [-dryrun] -host hostname [
login options]

polyspace-access -ver -host hostname [login options]
```

```
polyspace-access -encrypt-password
```

## Description

### Manage and View Projects

**Note** This Polyspace command is available in `*polyspaceroot*\polyspace\bin`. Here, *`polyspaceroot`* is the Polyspace Server installation folder, for instance, `C:\Program Files\Polyspace Server\R2023a` (see also “Installation Folder”). To avoid typing the full path to this command, add this location to the `PATH` environment variable in your operating system.

`polyspace-access -create-project folderPath -host hostname [login options]` creates a project folder in the Polyspace Access web interface. The folder can be at the top of the project hierarchy or a subfolder under an existing project folder.

`polyspace-access -move-project sourcePath -to-project-path destinationPath -host hostname [login options]` moves or renames a project or folder. The `sourcePath` and `destinationPath` must be absolute paths of the source and destination projects or folders. You cannot move or rename a project or folder if the path you specify for the destination already exists.

`polyspace-access -delete-project projectToDelete -host hostname [login options]` deletes specified project or folder from the Polyspace Access **Project Explorer**. The project or folder is moved to the **ProjectsWaitingForDeletion** folder, which is visible only to users with a role of **Administrator**. To completely delete the project or folder from the database, see “Delete Project Runs or Entire Projects”.

`polyspace-access -list-project [projectPath] -host hostname [login options]` without the optional argument `projectPath` lists the paths to all projects in the Polyspace Access database and their last run IDs. If you specify the full path to a project with the argument `projectPath`, the command lists the last run ID for only that project.

`polyspace-access -list-runs projectPath -host hostname [login options]` lists all the runs that were uploaded to the specified project. For each run, you see the run ID and any labels associated with that run.

### Upload, Download, or Export Results

`polyspace-access -upload pathToFileOrFolderOrZipFile [upload options] -host hostname [login options]` uploads Polyspace Bug Finder or Code Prover analysis results to the Polyspace Access database from the file (recommended), folder, or zipped file that you specify. You upload results using this command only if those results were generated with Polyspace Bug Finder Server or Polyspace Code Prover Server. You can upload results to an existing Polyspace Access project only if those results were generated by the same type of analysis. For instance, you cannot upload results of a Bug Finder analysis to a project that contains Code Prover results. To upload results generated with the Polyspace desktop interface, see “Upload Results from Polyspace Desktop Client”. Use the “upload options” on page 4-0 to specify a project parent folder or to rename a project.

`polyspace-access -export findingsToExport -output filePath [export options] -host hostname [login options]` exports project results from the Polyspace Access database to a text file whose location you specify with `filePath`. You can specify filters when you export the results or export a comparison between two projects. You specify the project using either the full path in Polyspace Access or the run ID. Use this command to export findings to other tools that you use for custom reports. To get the paths to projects and their last run IDs, use `polyspace-access` with the `-list-project` command.

`polyspace-access -download findingsToDownload -output-folder-path outputFolderPath -host hostname [login options]` downloads results from Polyspace Access project `findingsToDownload` to a folder whose location you specify with `outputFolderPath`. You specify the project using either the full path in Polyspace Access or the run ID.

You cannot open the downloaded results in the Polyspace desktop interface or the Polyspace as You Code IDE extensions. Use this command as part of the workflow to merge review information between projects, or to create a baseline for subsequent Polyspace analyses.

- To merge review information between projects, see “Import Review Information from Existing Polyspace Access Projects”.
- To create a baseline for Polyspace as You Code results at the command-line, see “Baseline Polyspace as You Code Results on Command Line”.

To get the paths to projects and their last run IDs, use `polyspace-access` with the `-list-project` command.

**Manage Review Information for Findings**

`polyspace-access -review fileOrFindingID -project-path projectToReview review options -host hostname [login options]` edits the information in the review fields of the findings in the project `projectToReview`. To review a single finding, pass a finding ID to the `-review` command . To perform a batch review, pass the path of a file where you store the finding IDs of multiple findings to the `-review` command. In this file, list one finding ID per line. You must specify at least one review field with `review options`, such as status, severity, comment, owner, or a bug tracking tool (BTT) ticket number.

You cannot assign a review field to a finding if that review field is set with a code annotation.

`polyspace-access -import-reviews sourceProjectPath -project-path targetProjectPath [-import-strategy never-overwrite-target | always-overwrite-target] -host hostname [login options]` imports review information from source project `sourceProjectPath` to target project `targetProjectPath`. Use this option if you have already reviewed findings in the source project and you reuse the code that contains those findings in the target project.

You can optionally specify one of these import strategies:

- `never-overwrite-target` (default) — If a review field in the target project already has content, do not overwrite that review field with the content from the source project
- `always-overwrite-target` — Always overwrite the content of the review fields in the target project with the content from the source project, even if the target review fields already have content.

See also “Import Review Information from Existing Polyspace Access Projects”.

**Manage Software Quality Objectives (SQO)**

`polyspace-access -set-sqo projectPathWithSQO -level sqoLevel [-name sqoName] -host hostname [login options]` assigns an SQO level to the specified project for the SQO definition that is currently applied to the project. You can optionally apply a different SQO definition to the project by specifying the name of that SQO definition with option `-name`. See also “Manage Software Quality Objectives in Polyspace Access”. You can run this command only on single projects but not on project folders.

`polyspace-access -get-sqo projectPathWithSQO -host hostname [login options]` shows the currently assigned SQO definition and SQO level for the specified project. See also “Manage Software Quality Objectives in Polyspace Access”. You can run this command only on single projects but not on project folders.

`polyspace-access -list-sqo -host hostname [login options]` lists the name of all the available SQO. See also “Manage Software Quality Objectives in Polyspace Access”.

#### Manage Project Run Labels and Unassigned Findings

`polyspace-access -add-label label -run-id runID -host hostname [login options]` adds a label to the project run specified with `runID`. Use labels to identify project runs of interest more easily, or to associate a run with a specific branch or continuous integration build job. See also “Manage Labels at the Command Line”.

`polyspace-access -remove-label label -run-id runID -host hostname [login options]` removes the specified label from the project run with run ID `runID`. If the specified label does not match any of the project run labels, the command is ignored. See also “Manage Labels at the Command Line”.

`polyspace-access -set-unassigned-findings findingsToAssign -owner userToAssign -source-contains pattern [set unassigned findings options] -host hostname [login options]` assigns owners to unassigned results in a project in the Polyspace Access database. You specify the project using either the full path in Polyspace Access or the run ID. Use the `set unassigned findings options` to assign findings from different source files or different groups of source files to different owners. To get the paths to projects and their last run IDs, use `polyspace-access` with the `-list-project` command.

#### Manage User Permissions

`polyspace-access -set-role role -user username -group groupname -project-path folderPathOrProjectPath -host hostname [login options]` assigns a role `role` to `username` or `groupname` for the specified project or project folder. A user or group role set for a project folder applies to all project findings under that folder. All users in a group inherit the role assigned to their parent group. You specify the project using either the full path in Polyspace Access or the last run ID. To get the paths to projects and their last run IDs, use `polyspace-access` command with the `-list-project` command.

`polyspace-access -unset-role -user username -group groupname -project-path folderPathOrProjectPath -host hostname [login options]` removes any role previously assigned to `username` or `groupname` for the specified project or project folder. Unassigning a role for a group unassigns that role for all the users in that group. You specify the project using either the full path in Polyspace Access or the last run ID. To get the paths to projects and their last run IDs, use `polyspace-access` with the `-list-project` command.

#### Migrate Results from Polyspace Web Metrics

`polyspace-access -generate-migration-commands metrics_dir -output-folder-path dir [generate migration commands options]` generates scripts to migrate projects from the path `metrics_dir` in Polyspace Metrics to Polyspace Access. The command stores the scripts in `dir`. To specify which project findings to migrate, use `generate migration commands options`.

`polyspace-access -migrate -option-file-path dir [-dryrun] -host hostname [ login options]` migrates projects from Polyspace Metrics to Polyspace Access using the scripts generated with the `-generate-migration-commands` command. To view which projects are migrated without actually migrating the projects, use the `-dryrun` option.

**Other Commands**

`polyspace-access -ver -host hostname [login options]` displays the version and license number of the Polyspace Access instance that you specify with `hostname`, and the version of the `polyspace-access` binary. You can run this command only with Polyspace Access instances for which you have login credentials.

`polyspace-access -encrypt-password` encrypts the password you use to log into Polyspace Access. Use the output of this command as the argument of the `-encrypted-password` option when you write automation scripts to interact with Polyspace Access.

## Examples

### Encrypt Password and Store Login Options in a Variable

Polyspace Access requires login credentials. You can enter the credentials at the command-line when you execute a command, or you can generate an encrypted password to use with other login options in automation scripts.

To encrypt your password, use the `-encrypt-password` command and enter your Polyspace Access credentials. The command uses the user name and password you enter to generate an encrypted password.

```
polyspace-access -encrypt-password
login: jsmith
password:
CRYPTED_PASSWORD KEAGKAMJMCOPLFKPKOHOJNDJCBACFJBL
Command Completed
```

Store the login and encrypted password in a credentials file and restrict read and write permission on this file. Open a text editor, copy these two lines in the editor, then save the file as `myCredentials.txt` for example.

```
 -login jsmith
 -encrypted-password KEAGKAMJMCOPLFKPKOHOJNDJCBACFJBL
```

To restrict the file permissions, right-click the file and select the **Permissions** tab on Linux or the **Security** tab on Windows.

If you manage your analysis findings through automated scripts, create a variable to store the connection configuration and login credentials. Use this variable in your script, or at the command line to avoid typing your credentials when you execute a command.

```
set LOGIN=-host myAccessServer -port 1234 -credentials-file myCredentials.txt
```

```
polyspace-access %LOGIN% -create-project myProject
```

### Create a Project Folder with Restricted Access and Upload to Folder

Suppose that you want to upload a set of findings to Polyspace Access generated with Polyspace Bug Finder Server or Polyspace Code Prover Server, and authorize only some team members to view these findings.

Create a project folder `restrictedProject` at the top of the project hierarchy.

```
polyspace-access -host myAccessServer -port 1234 ^
-create-project restrictedProject
```

Set roles for users `jsmith` and `rroll`, and group `Authorized Users`, authorizing them to access the project folder as contributors. If a user name or group name contains spaces, enclose it in double quotes.

```
polyspace-access -host myAccessServer ^
-port 1234 -set-role contributor ^
-user jsmith -user rroll -group "Authorized Users" -project-path restrictedProject
```

Aside from the creator of the project folder and the users and group with a role of contributor, no other user can view or access any findings uploaded to `restrictedProject`.

Upload Bug Finder project findings from project "My Example Project" to parent folder `Restricted`.

```
polyspace-access -host myAccessServer -port 1234 ^
-upload "C:\Polyspace_Workspace\My Example Project\Module1\ps_results.psbf" ^
-parent-project Restricted
```

The uploaded findings are stored under `restrictedProject/My_Example_Project (Bug Finder)`. The `-upload` command outputs information about the upload including an `ACCESS URL`. You can use that URL to view the uploaded results in the Polyspace Access interface.

See also “Manage Project Permissions”.

### Assign Results to Component Owners and Export Assigned Results

If you follow a component-based development approach, you can assign analysis findings by component to their respective owners.

Get a list of projects currently stored on the Polyspace Access database. The command outputs a list of project findings paths and their last run ID.

```
polyspace-access -host myAccessServer -list-project

Connecting to https://example-access-server:9443
Connecting as jsmith

Get project list with the last Run Id
Restricted/Code_Prover_Example (Code Prover) RUN_ID 14
multimodule/vxWorks_demo (Code Prover) RUN_ID 16
public/Bug_Finder_Example (Bug Finder) RUN_ID 24
public/CP/Code_Prover_Example (Polyspace Code Prover) RUN_ID 8
public/Polyspace (Code Prover) RUN_ID 28
Command Completed
```

Assign all red and orange run-time error findings to the owner of all the files in `Component_A` of project `vxWorks_demo`. Perform the same assignment for the owner of `Component_B`. To specify the `vxWorks_demo` project, use the run ID.

```
polyspace-access -host myAccessServer ^
-set-unassigned-findings 16 ^
-owner A_owner -source-contains Component_A ^
```

```
-owner B_owner -source-contains Component_B ^
-rte Red -rte Orange
```

`-source-contains Component_A` matches all files with a file path that contains `Component_A`.

`-source-contains Component_B` matches all files with a file path that contains `Component_B`, but excludes files with a file path that contains `Component_A`.

After you assign findings, export the findings and generate `.csv` files for each owner containing the findings assigned to them.

```
polyspace-access -host myAccessServer ^
-export 16 ^
-output C:\Polyspace_Workspace\myResults.csv ^
-output-per-owner
```

The command generates file `myResults.csv` containing all findings from the project with run ID 16. The command also generates files `myResutls.csv.A_owner.csv` and `myResults.csv.B_owner.csv` on the same file path.

### Make Batch Edits to Review Information for New Findings

If you want to make batch edits to the review information of analysis results after you upload those results to Polyspace Access:

1. Export a list of findings that you want to review.
2. Edit the review information for each finding using the finding IDs stored in the file from step 1.

Export a list of findings that you want to review, using the `polyspace-access -export` command. For example, if you want to export all new defects with severity **High** or **Medium** from a project with project path `public/example/Bug_Finder_Example (Bug Finder)` and project ID 129, run this command:

```
polyspace-access $login -export 129 -output newMedHighDefects.txt\
 -resolution new -defects High -defects Medium
```

The command outputs the tab separated values (TSV) file `newMedHighDefects.txt`. The file contains a list of findings that match the filters that you specified, where the first column of the file shows the finding IDs.

Here, `$login` is a variable that stores the login credentials and other connection information. To configure this variable, see “Encrypt Password and Store Login Options in a Variable” on page 4-10.

Extract the finding IDs column from the file that you generated in the previous step to another text file. For instance, you use the `awk` command (BASH command) or a `for` loop (DOS command) to skip the column header and then save the finding IDs to file `findingIDs.txt`.

```
#BASH COMMAND
awk ' NR>1 {print$1} ' newMedHighDefects.txt >> findingIDs.txt

##########################################

REM DOS COMMAND
for /f "skip=1" %i in (newMedHighDefects.txt); do @echo %i >> findingIDs.txt
```

Pass the file `findingIDs.txt` to the `polyspace-access -review` command. Specify the path of the project that contains those findings and the review fields that you want to edit. For instance, you might want to assign a status, owner, and comment to the findings.

```
# BASH COMMAND

 polyspace-access $login -review findingIDs.txt \
 -project-path "public/example/Bug_Finder_Example (Bug Finder)" \
 -set-status "To investigate" -set-owner jsmith \
 -set-comment "Finding assigned automatically"

######################################################

REM DOS COMMAND

polyspace-access %login% -review findingIDs.txt ^
 -project-path "public/example/Bug_Finder_Example (Bug Finder)" ^
 -set-status To investigate" -set-owner jsmith ^
 -set-comment "Finding assigned automatically"
```

The command runs once and performs a batch edit of the specified review fields for all the findings listed in the file `findingsIDs.txt`. Depending on the number of findings and on your network speed, the command might take a few moments to complete.

### Migrate Projects from Metrics to Polyspace Access

If you have projects stored on a Polyspace Metrics server, you can migrate them to the Polyspace Access database. Log in to your Metrics server to complete this operation.

Generate migration scripts for the projects you want to migrate. Specify the folder path of the location where the projects are stored, for example `C:\Users\jsmith\AppData\Roaming\Polyspace_RLDatas\results-repository`

```
polyspace-access -generate-migration-commands ^
C:\Users\jsmith\AppData\Roaming\Polyspace_RLDatas\results-repository ^
-output-folder-path C:\Polyspace_Workspace\toMigrate -project-date-after 2017-06
```

The command generates migration scripts for all projects in the specified metrics folder that were uploaded on or after June 2017. The scripts are stored in folder `C:\Polyspace_Workspace\toMigrate`.

Use the `-dryrun` option to check which projects will be migrated.

```
polyspace-access -host myAccessServer ^
-migrate -option-file-path ^
C:\Polyspace_Workspace\toMigrate -dryrun
```

The command output contains a list of projects. Inspect it to ensure that you are migrating the correct projects.

To perform the migration, rerun the last command without the `-dryrun` option.

## Input Arguments

#### Connection Configuration and Login

### `hostname` — Polyspace Access machine host name

string

`hostname` corresponds to the host name that you specify in the URL of the Polyspace Access interface, for example `https://hostname:port/metrics/index.html`. If you are unsure about which host name to use, contact your Polyspace Access administrator. The default host name is `localhost`. You must specify a host name with all `polyspace-access` commands, except the `-generate-migration-commands` and `-encrypt-password` commands .

Example: `-host myAccessServer`

### `login options` — Options to configure connection to Polyspace Access

string

Options to specify connection configuration and login credentials.

**Connection Options**

| Option | Description |
|---|---|
| `-port portNumber` | `portNumber` corresponds to the port number that you specify in the URL of the Polyspace Access interface, for example `https://hostname:portNumber/metrics/index.html`. If you are unsure about which port number to use, contact your Polyspace Access administrator. The default port number is `9443`. |
| `-protocol http \| https` | HTTP protocol used to access Polyspace Access. The default protocol is `https`. |
| `-credentials-file filePath` | Full path to the text file where you store your login credentials. Use this option if, for instance, you use a command that requires your Polyspace Access credentials in a script but you do not want to store your credentials in that script. While the script runs, someone inspecting currently running processes cannot see your credentials.<br>You can store only one set of credentials in the file, either as `-login` and `-encrypted-password` entries on separate lines, for instance:<br>`-login jsmith`<br>`-encrypted-password LAMMMEACDMKEFELKMNDCONEAPECEEKPL`<br>or as an `-api-key` entry:<br>`-api-key keyValue123`<br>Make sure that you restrict the read and write permissions on the file where you store your credentials. For example, to restrict read and write permissions on file `login.txt` in Linux, use this command:<br>`chmod go-rwx login.txt` |

| Option | Description |
|---|---|
| `-api-key` *`keyValue`* | API key you use as a login credential instead of providing your login and encrypted password. To assign an API key to a user, see “Configure User Manager” or contact your Polyspace Access administrator.<br>Use the API key if, for instance, you use a command that requires your Polyspace Access login credentials as part of an automation script with a CI tool like Jenkins. If a user updates his or her password, you do not need to update the API key associated with that user in your scripts.<br>It is recommended that you store the API key in a text file and pass that file to the command by using `-credentials-file`. See the description for option `-credentials-file`. |
| `-login` *`username`*<br>`-encryted-password` *`ENCRYPTED_PASSWD`* | Login credentials you use to interact with Polyspace Access. The argument of `-encrypted-password` is the output of the `-encrypt-password` command.<br>If you do not use these two options, you are prompted to enter your credentials at the command line, unless you use `-api-key`. |
| `-max-retry` *`retryNumber`* | Number of times the `polyspace-access` command retries to upload results when you upload from a client machine to the server machine that hosts Polyspace Access. Specify this option to retry the upload command in the event of sporadic network outages. The command waits 10 seconds between retries.<br>By default, the command retries 3 times. |

**Miscellaneous Options**

| Option | Description |
|---|---|
| `-output` *`filePath`* | Full path to the file where you store command outputs. |
| `-tmp-dir` *`folderPath`* | Folder path where you store temporary files generated by the `polyspace-access` commands. The default folder path is `tmp/ps_results_server` on Linux and `C:/Users/%username%/AppData/Local/Temp/ps_results_server` on Windows. |
| `-log` *`filePath`* | File path where you store the command output log. By default the command does not generate a log file. |
| `-h` | Display the help information for `polyspace-access` or one of its commands. |

### Manage and View Projects

#### `folderPath` — Absolute path of folder

string

Absolute Project folder path specified as a string. If the name includes spaces, use double quotes. Specify the full path to folders nested under a parent folder.

If your folder path involves a folder that does not already exist, the folder is created.

Example: `-create-project topFolder`

Example: `-create-project "topFolder/subFolder/Folder has spaces"`

#### `sourcePath` — Absolute path of source project or folder
string

Absolute path of project or folder specified as a string. If the name includes spaces, use double quotes. Specify the absolute path of the folder or project that you want to move or rename.

Example: `-move-project old/Folder/Path`

Example: `-move-project "old/Folder/name has spaces"`

#### `destinationPath` — Absolute path of source project or folder
string

Absolute path of project or folder specified as a string. If the name includes spaces, use double quotes. Specify the absolute path of the new folder or project location.

If you move or rename a folder, all subfolders and subprojects are moved to the new parent folder. If you move a project to a different folder, you must specify the absolute path including the project name for the destination path.

Example: `-move-project new/Folder/Path`

Example: `-move-project "new/Folder/name has spaces"`

#### `projectToDelete` — Absolute path of project or folder
string

Absolute path of project or folder specified as a string. If the name includes spaces, use double quotes.

Specify the absolute path of the folder or project that you want to delete. If you delete a folder, all subfolders and subprojects under that folder are also deleted.

Example: `-delete-project public/Folder/projectName`

Example: `-delete-project "public/Folder name has spaces"`

#### `projectPath` — Absolute path of project
string

Absolute path of the project findings. Specify this optional argument with `-list-project` to get the path and the last run ID of the corresponding project, or with `-list-runs` to get run ID and labels of all the runs that you uploaded to the project.

If the path name includes spaces, use double quotes.

Example: `-list-project "public/Examples/Bug_Finder_Example (Bug Finder)"`

Example: `-list-runs public/Examples/Code_Prover_Example`

### Upload, Download, or Export Results

#### `pathToFileOrFolderOrZipFile` — Path to file, folder, or zipped file containing analysis results
string

Path of a Polyspace Bug Finder (PSBF) or Code Prover (PSCP) analysis results file specified as a string. The command uploads copies of the results file and other analysis results files located in the same folder. Alternatively, you can specify the path of a parent folder that contains a single PSBF or PSCP results file, or the path of a zipped results file or folder.

For instance, for the Bug Finder results stored in `C:\Polyspace_Workspace\myProject\Module_1\BF_results\ps_results.psbf`, specify this with the `-upload` command. If the path name includes spaces, enclose the path in double quotes.

For faster uploads, store your analysis results in a dedicated results folder by using option `-results-dir` when you run the analysis. If you store results in a folder that contains a large number of files unrelated to Polyspace analysis results, for example the root folder of your repository, Polyspace Access takes longer to upload the results.

Example: `-upload C:\Polyspace_Workspace\myProject\Module_1\BF_results\ps_results.psbf`

Example: `-upload C:\Polyspace_Workspace\myProject\Module_1\ -project projectFolder`

### `upload options` — Options to specify where to upload results

string

Options to specify path to project folder and name of project where you upload results. These options are not optional.

| Option | Description |
|---|---|
| `-parent-project` *`folderPath`* | Path of the parent folder in the Polyspace Access explorer under which you store uploaded results.<br>If you do not specify a parent folder, uploaded results are stored under the **public** folder. |
| `-project` *`resultsName`* | Name of the uploaded results in the Polyspace Access explorer. Use this option to rename the results that you upload.<br>If you do not specify a results name, the results are stored as `projectName (resultsType)`, where spaces in the `projectName` are replaced with underscores. For instance, if you upload Bug Finder results from project Bug Finder Example, these results are stored as `Bug_Finder_Example (Bug Finder)`. |

### `findingsToExport` — Project findings path or run ID

string

Path or run ID of the project findings that you export. Polyspace assigns a unique run ID to each analysis run you upload. If the path name includes spaces, use double quotes. To get the project findings path or last run ID, use `-list-project`.

Example: `-export "public/Examples/Bug_Finder_Example (Bug Finder)"`

Example: `-export 4`

### `filePath` — Path to file containing command output

string

Path to the file that stores the output of the command when you specify the `-output` option. This option is mandatory with the `-export` command.

Example: `-output C:\Polyspace_Workspace\myResults.txt`

### `export options` — Options to specify which findings to export

string

Options to specify where to export findings, and which subset of findings you export. Use these options to export findings to other tools you use to create custom reports or other custom review templates.

| Option | Description |
|---|---|
| `-output` *`filePath`* | File path where you export the findings. This option is mandatory with the `-export` command. |
| `-resolution New` | Export only new findings compared to the previous analysis (previous upload with the same project name). If the project contains only one run, the command errors out. When there is only one project run, all findings are new and you do not need to specify this option. |
| `-output-per-owner` | Use this option to generate files that only contain findings assigned to a particular user. The files are stored on the path you specify with `-output`.<br>For an example of how to assign findings to owners at the command-line, see “Assign Results to Component Owners and Export Assigned Results” on page 4-11. |
| `-rte` *`All`* `\|` *`Red`* `\|` *`Gray`* `\|` *`Orange`* `\|` *`Green`* | Type of RTE finding to export. Specify `All`, `Red`, `Gray`, `Orange`, or `Green`.<br>To specify more than one argument, call the option for each argument. For example, `-rte Red -rte Orange`. |
| `-defects` *`All`* `\|` *`High`* `\|` *`Medium`* `\|` *`Low`* | Impact of DEFECTS findings to export. Specify `All`, `High`, `Medium`, or `Low`.<br>To specify more than one argument, call the option for each argument. For example, `-defects Medium -defects Low`. |
| `-custom-coding-rules` | Export all custom coding rules findings. |
| `-coding-rules` | Export all coding rules findings. |
| `-code-metrics` | Export all code metrics findings. |
| `-global-variables` | Export all global variables findings. |
| `-status` *`Unreviewed`* `\|` `"`*`To investigate`*`"` `\|` `"`*`To fix`*`"` `\|` *`Justified`* `\|` `"`*`No action planned`*`"` `\|` `"`*`Not a defect`*`"` `\|` *`Other`* | Review status of the findings to export. Specify `Unreviewed` (default), `"To investigate"`, `"To fix"`, `Justified`, `"No action planned"`, `"Not a defect"`, or `Other`.<br>To specify more than one argument, call the option for each argument. For example, `-status Tofix -status Toinvestigate`. |
| `-severity` *`All`* `\|` *`High`* `\|` *`Medium`* `\|` *`Low`* | Severity of the findings to export. Specify `All`, `High`, `Medium`, or `Low`.<br>To specify more than one argument, call the option for each argument. For example, `-severity High -severity Low`. |
| `-unassigned-findings` | Export all unassigned findings. |

| Option | Description |
|---|---|
| `-open-findings-for-sqo` *1 \| 2 \| 3 \| 4 \|5 \| 6* | Software quality objective or SQO level that must be satisfied. Specify a number from 1 to 6 for *`sqo_level`*. If you specify an SQO level, the `polyspace-access` command exports only open findings that must be fixed or justified to satisfy the requirements of this level.<br>The SQO levels 1 to 6 specify an increasingly stricter set of predefined or user-defined requirements in terms of Polyspace results. To customize the requirements in the Polyspace Access web interface, see “Customize Software Quality Objectives”<br>For more information on the SQO levels, see “Evaluate Polyspace Bug Finder Results Against Bug Finder Quality Objectives”.<br>For instance, SQO level 2 in Code Prover requires that you must not have unjustified red checks. This specification means that if you use `-open-findings-for-sqo` with a level higher than 2, all red checks are exported and must be subsequently fixed or justified. If you want to impose this requirement in the earlier SQO level 1, you can customize level 1 in the Polyspace Access web interface. |
| `-resolution` *New \| Fixed \| Unresolved \| Resolved* | Use this option to compare two runs from the same project. The resolution corresponds to the type of comparison that the software performs between the current run and a baseline run that you specify with option `-baseline`. The baseline run must be older than the current run.<br>The resolution types are:<br>• `New` — Findings that are in the current run but not in the baseline run.<br>• `Fixed` — Findings that are fixed in the current run, either because the finding was fixed in the source code, or because the source code containing the finding is deleted or no longer part of the analysis.<br>• `Unresolved` — Findings from the baseline run that are still present in the current run.<br>• `Resolved` — Findings that are `Fixed` in the current run or findings with a status of `Justified`, `No Action Planned`, or `Not A Defect` in the current run.<br>If you do not specify a baseline, the current run is compared to the run that immediately precedes it. If the project contains only one run, you get an error.<br>You can specify only one resolution type at a time. To export a comparison for multiple resolution types, run the `polyspace-access -export` command for each resolution type.<br>See also “Comparison Mode at the Command Line”. |

| Option | Description |
|---|---|
| `-baseline` *`runID`* | Specify the run ID of a run that you use as a baseline for comparison with a current run. This option requires option `-resolution`.<br>This option is not mandatory. If you do not specify a baseline, the current run is compared to the run that immediately precedes it. If the project contains only one run, you get an error.<br>See also “Comparison Mode at the Command Line”. |
| `-imported-reviews` *`Not applied \| Written \| Overwritten \| Unconfirmed`* | Use this option to export a list of findings from a project (target) whose review information was imported from another project (source). Specify one of these types of imports:<br>• `Not applied` — Review information was imported from the source project but the review fields in the target project kept the original values.<br>• `Written` — The Review information from the source project was written to the target project only if the review fields in the target project were previously empty.<br>• `Overwritten` — The Review information from the source project was written to the target project even if the review fields in the target project were not previously empty.<br>• `Unconfirmed` — Use this filter to select findings that where the result of the import operation has not been confirmed by a reviewer. You confirm the result of the import operation in the Polyspace Access interface. See “Confirm Imported Review Information” .<br>See also “Import Review Information from Existing Polyspace Access Projects” |

You can also use a combination of options. For instance, `-coding-rules -severity High` exports coding rule violations that have been assigned a status of `High` in the Polyspace Access web interface.

### `findingsToDownload` — Project findings path or run ID

string

Path or run ID of the project findings that you download. Polyspace assigns a unique run ID to each analysis run that you upload to Polyspace Access. If the path name includes spaces, use double quotes. To get the project findings path or latest run ID, use `-list-project`.

When you specify the project path, the command downloads the latest run of that project. To download an older run, specify the run ID. To obtain the run ID of older runs, in the Polyspace Access interface, select a project in the **Project Explorer**, and then click the **Current** drop-down selection in the toolstrip to view the available run IDs.

Example: `-download "public/Examples/Bug_Finder_Example (Bug Finder)"`

Example: `-download 5113`

### `outputFolderPath` — Path to folder containing downloaded results

string

Path of the folder where you store the downloaded results. If the folder you specify already exists, it must be empty. This option is mandatory with the `-download` command.

Example: `-output-folder-path C:\Polyspace_Workspace\baseline`

#### Manage Review Information for Findings

### `fileOrFindingID` — finding ID or list of finding IDs to review

integer

Specify the ID of the finding to which you want to add review information by using the command `-review`. To perform a batch review, store the finding IDs of multiple findings in a text file (one finding ID per line) and pass that file to `-review` instead.

Polyspace Access assigns unique IDs to each finding that you upload.

Example: `-review 1234`

Example: `-review listOfFindings.txt`

### `projectToReview` — Absolute path of project

string

Specify the absolute path of the project that contains the findings to which you want to add review information. Use option `-project-path` to specify the path. When you use `-review`, this option is mandatory .

Example: `-project-path path/To/Project`

Example: `-project-path "project/path/with spaces"`

### `review options` — Options to specify review fields

string

Options to assign one or more review fields to the finding with the specified finding ID. When you use `polyspace-access -review`, you must specify at least one review option . After you assign review information to the finding, that information is available in the **Result Details** pane in Polyspace Access.

| Option | Description |
|---|---|
| `-set-status` *`Unreviewed \| "To investigate" \| "To fix" \| Justified \| "No action planned" \| "Not a defect" \| Other`* | Assign a status of the finding. Specify `Unreviewed` (default), `"To investigate"`, `"To fix"`, `Justified`, `"No action planned"`, `"Not a defect"`, or `Other`.<br>To reset a finding status, set the status to `Unreviewed`. |
| `-set-severity` *`Unset \| High \| Medium \| Low`* | Assign severity of the finding. Specify `Unset` (default), `High`, `Medium`, or `Low`.<br>To reset a finding severity, set the severity to `Unset`. |
| `-set-comment` *`comment`* | Assign a comment to the finding. |
| `-unset-comment` | Delete the comment currently assigned to the finding. |

| Option | Description |
|---|---|
| `-set-owner` *`username`* | Assign an owner to the finding. Specify the username that the user uses to log into Polyspace Access, not the display name. For instance, specify `jsmith` instead of `John Smith`. |
| `-unset-owner` | Remove the owner currently assigned to the finding. |
| `-set-ticket` *`ticketID`* | Assign an existing BTT ticket to the finding. Specify the ticket ID.<br>You can assign a BTT ticket to a finding only if you integrate a BTT such as Jira Software or Redmine with Polyspace Access. Contact your Polyspace administrator to determine if Polyspace is configured for issue tracking. |
| `-unset-ticket` | Remove the BTT ticket currently applied to the finding. |

**`sourceProjectPath` — Absolute path of project**
string

Specify the absolute path of the project where you have already reviewed the findings. You import the review information from this source project.

Example: `-import-reviews path/To/sourceProject`

Example: `-import-reviews "source project/path/with spaces"`

**`targetProjectPath` — Absolute path of project**
string

Specify the absolute path of the target project to which you import the review information. By default, if a review field already has content, the imported review information from the source project does not overwrite it. Use option `-import-strategy always-overwrite-target` to force the overwrite of the target project review information with the review information imported from the source project.

Example: `-to-project-path path/To/targetProject`

Example: `-import-reviews "target project/path/with spaces"`

**Manage Software Quality Objectives (SQO)**

**`projectPathWithSQO` — Absolute path of project**
string

Specify the absolute path of a project with the command `-set-sqo` to assign an SQO level and optionally an SQO definition. To view the currently applied SQO level and SQO definition for the specified project, use the command `-get-sqo`. You cannot specify a folder path with the commands `-set-sqo` and `-get-sqo`.

If your project path involves a folder that does not already exist, the folder is created.

Example: `-set-sqo path/To/Project`

Example: `-get-sqo "project/path/with spaces"`

**`sqoLevel` — SQO Level to assign**
`1 | 2 | 3 | 4 | 5 | 6 | exhaustive`

Level of the SQO to assign to the specified project for the currently applied SQO definition. Use option `-name` to also assign a different SQO definition.

Example: `-level 3`

Example: `-level exhaustive`

### `sqoName` — Name of SQO definition to assign

string

Specify an SQO definition name (optional) to apply a different SQO definition to the specified project.

To see a lit of available SQO definitions, use the command `-list-sqo`. To view the currently applied SQO definition for the specified project, use the command `-get-sqo`.

If the SQO definition name contains spaces, use double quotes.

Example: `-name Polyspace`

Example: `-name "My Custom SQO"`

#### Manage Project Run Labels and Unassigned Findings

### `label` — Project run label to add or remove

string

Specify a label to add or to remove from the project run that you specify with option `-run-id`. If the label includes spaces, use double quotes.

Example: `-add-label testing_branch`

Example: `-remove-label "testing branch"`

### `runID` — run ID of project run

string

Specify the run ID of the project run to which you add a label or from which you remove a label. To view the latest run IDs for a project, use the command `-list-project`. To view the run IDs of all the project runs for a specific project, use the command `-list-runs`.

Example: `-run-id 1234`

### `findingsToAssign` — Project findings path or run ID

string

Path or run ID of the project findings that you assign to a user. Polyspace assigns a unique run ID to each analysis run you upload. If the path name includes spaces, use double quotes. To get the project findings path or last run ID, use `-list-project`.

Example: `-set-unassigned-findings "public/Examples/Bug_Finder_Example (Bug Finder)"`

Example: `-set-unassigned-findings 4`

### `userToAssign` — Polyspace Access user name

string

User name of user you assign as owner of unassigned findings. To assign multiple owners, call the option for each user.

Each call to `-owner` must be paired with a call to `-source-contains`.

Example: `-owner jsmith`

### `pattern` — Pattern to match against file path

string

Pattern to match against file path of project source files. To match file paths for all source files, use "`-source-contains /`".

Enter a substring from the file path. You cannot use regular expressions.

When you call this option more than once, each instance excludes patterns from previous instances. For example, `-source-contains foo -source-contains bar` matches all file paths that contain `foo`, then all file paths that contain `bar` excluding paths that contain `foo`.

When you assign findings to multiple owners, call this option for each call to `-owner`.

Example: `-source-contains main`

### `set unassigned findings options` — Options to specify which findings to assign

string

Options to assign all findings or only a subset based on component or individual source files. To make an assignment, specify a pattern to match against the folder or file paths to assign.

| Option | Description |
|---|---|
| `-rte` *`All \| Red \| Gray \| Orange \| Green`* | Type of unassigned RTE finding to assign. Specify `All`, `Red`, `Gray`, `Orange`, or `Green`.<br>To specify more than one argument, call the option for each argument. For example, `-rte Red -rte Orange`. |
| `-defects` *`All \| High \| Medium \| Low`* | Impact of unassigned DEFECTS findings to assign. Specify `All`, `High`, `Medium`, or `Low`.<br>To specify more than one argument, call the option for each argument. For example, `-defects Medium -defects Low`. |
| `-custom-coding-rules` | Assign owner to all unassigned custom coding rules findings. |
| `-coding-rules` | Assign owner to all unassigned coding rules findings. |
| `-code-metrics` | Assign owner to all unassigned code metrics findings. |
| `-global-variables` | Assign owner to all unassigned global variables findings. |
| `-resolution New` | Assign owner to unassigned findings that are new compared to previous analysis run. If the project contains only one run, the command errors out. When there is only one project run, all findings are new and you do not need to specify this option. |
| `-status` *`Unreviewed \| "To investigate" \| "To fix" \| Justified \| "No action planned" \| "Not a defect" \| Other`* | Review status of the unassigned findings to assign. Specify `Unreviewed`, `"To investigate"`, `"To fix"`, `Justified`, `"No actionp lanned"`, `"Not a defect"`, or `Other`.<br>To specify more than one argument, call the option for each argument. For example, `-status "To fix" -status "To investigate"`. |

| Option | Description |
|---|---|
| `-severity` *`All \| High \| Medium \| Low`* | Severity of the unassigned findings to assign. Specify `All`, `High`, `Medium`, or `Low`.<br>To specify more than one argument, call the option for each argument. For example, `-severity High -severity Low`. |
| `-dryrun` | Display command output without making any assignment. Use this option to check that your assignments are correct. |

#### Manage User Permissions

### `role` — Level of access permissions for project folder or findings

`owner | contributor | forbidden`

Level of access to project folder and findings for a user.

- **owner**: User can move, rename, or delete specified project folders or findings and review their content.
- **contributor**: User can review content of specified project folder or findings.
- **forbidden**: User cannot access specified project folder or findings. Set this role to restrict the access of a user to a set of project findings inside a project folder that is accessible to the user.

Example: `-set-role contributor`

### `username` — Polyspace Access user name

string

Polyspace Access user name.

Example: `-user jsmith`

### `groupname` — Polyspace Access group name

string

Polyspace Access group name.

Example: `-group "Group UI team"`

### `folderPathOrProjectPath` — Project folder or findings path

string

Path of a project folder or project findings. When `folderPathOrProjectPath` is the path to a project folder, the user role you set applies to all subfolders and project findings under that folder. If the path name includes spaces, use double quotes. To get the project folder or findings path, use `-list-project`.

Example: `-project-path "public/Examples/Bug_Finder_Example (Bug Finder)"`

Example: `-project-path public`

#### Migrate Results from Metrics to Polyspace Access

### `metrics_dir` — Folder path of Polyspace Metrics projects

string

Path of folder containing the Polyspace Metrics projects you want to migrate to Polyspace Access.

Example: `-generate-migration-commands C:\Users\%username%\AppData\Roaming\Polyspace_RLDatas\results-repository`

**`dir` — Output folder for migration scripts**
string

Path to folder that stores the output of `-generate-migration-commands`. Do not specify an existing folder.

Example: `local/Polyspace_Workspace/migration_scripts`

**`generate migration commands options` — Options to specify which projects to migrate**
string

| Option | Description |
|---|---|
| `-output-folder-path` *`folderPath`* | Folder path where you want to store the generated command files. Do not specify an existing folder. |
| `-max-project-runs` *`int`* | Number of most recent analysis runs you want to migrate for each project. For instance, to migrate only the last two analysis runs of a project, specify 2. |
| `-project-date-after` *`YYYY[-MM[-DD]]`* | Only migrate results that were uploaded to Polyspace Metrics on or after the specified date. |
| `-product` *`productName`* | Product used to analyze and produce project findings, specified as `bug-finder`, `code-prover`, or `polyspace-ada`. |
| `-analysis-mode` *`integration \| unit-by-unit`* | Analysis mode use to generate project findings, specified as `integration` or `unit-by-unit`. |

## Version History

**Introduced in R2019a**

### See Also

**Topics**
"Run Polyspace Bug Finder on Server and Upload Results to Web Interface"
"Send Email Notifications with Polyspace Bug Finder Server Results"
"Baseline Polyspace as You Code Results on Command Line"

# polyspace-bug-finder-access

(DOS/UNIX) Run Polyspace as You Code from Windows, Linux, or other command line

## Syntax

```
polyspace-bug-finder-access -sources sourceFile [options]
```

```
polyspace-bug-finder-access -sources sourceFile -import-comments
baselineFolder [options]
```

```
polyspace-bug-finder-access -sources sourceFile -options-file optFile
```

```
polyspace-bug-finder-access -h[elp]
```

## Description

**Note** This Polyspace command is available in `*polyspaceroot*\polyspace\bin`. Here, *`polyspaceroot`* is the Polyspace as You Code installation folder, for instance, `C:\Program Files\Polyspace as You Code\R2023a` (see also “Run Polyspace as You Code from Command Line and Export Results”). To avoid typing the full path to this command, add this location to the `PATH` environment variable in your operating system.

`polyspace-bug-finder-access -sources sourceFile [options]` runs a Polyspace as You Code analysis on the source file `sourceFile`. You can customize the analysis with additional options.

`polyspace-bug-finder-access -sources sourceFile -import-comments baselineFolder [options]` runs a Polyspace as You Code analysis on the source file `sourceFile` and then imports the review information from a previous run stored in `baselineFolder`. See `-import-comments`.

Use this workflow to compare your results against the results of a previous run that you download from Polyspace Access and focus on new results only or on unreviewed results. See “Baseline Polyspace as You Code Results on Command Line”.

`polyspace-bug-finder-access -sources sourceFile -options-file optFile` runs a Polyspace as You Code analysis on the source file `sourceFile` with the options specified in the option file. When you have many analysis options, an options file makes it easier to run the same analysis again.

`polyspace-bug-finder-access -h[elp]` lists a summary of possible analysis options.

## Examples

### Run Analysis by Directly Specifying Options

Run the `polyspace-bug-finder-access` command by specifying all options directly at the command line.

Run the `polyspace-bug-finder-access` command on a single source file `file.c` in the current folder. Save the results in the folder `polyspaceResults`.

```
polyspace-bug-finder-access -sources file.c -results-dir polyspaceResults
```

### Run Analysis Using Options File

Run the `polyspace-bug-finder-access` command by specifying options in a text file.

Enter the following in a text file `opts.txt`:

```
-results-dir polyspaceResults
-compiler gnu7.x
```

Run the `polyspace-bug-finder-access` command on a single source file `file.c` in the current folder. Use the analysis options in the previously created file `opts.txt`:

```
polyspace-bug-finder-access -sources file.c -options-file opts.txt
```

## Input Arguments

### `sourceFile` — C or C++ file to analyze

source file name or path

C or C++ source file name, specified as a string. If the file is not in the current folder (`pwd`), `sourceFile` must include a full or relative path. To avoid errors because of paths with spaces, add quotes `" "` around the path. For more information, see `-sources`.

Note that you can only analyze a single source file at a time using Polyspace as You Code.

Example: `myFile.c`

### `options` — Analysis option and corresponding value

command-line flag with optional value

Analysis options and their corresponding values, specified by the option name and if applicable value. For syntax specifications, see the individual analysis option reference pages.

Example: `-lang C-CPP` `-compiler diab`

### `optFile` — Text file listing analysis options and values

options file name or path

Text file listing analysis options and values, specified as a string. If the file is not in the current folder (`pwd`), `optFile` must include a full or relative path. To avoid errors because of paths with spaces, add quotes `" "` around the path. For more information, see `-options-file`.

Example: `opts.txt`, `"C:\ps_analysis\options.txt"`

### `baselineFolder` — Folder where you download baseline run

folder path

Path of folder where you store the run that you download from Polyspace Access. You use the downloaded run as a baseline for Polyspace as You Code runs. See “Baseline Polyspace as You Code Results on Command Line”.

Example: `"C:\Temp\Results_Folder\baseline"`

## Version History

**Introduced in R2021a**

## See Also

`polyspace-results-export` | `polyspace-configure`

### Topics

"Run Polyspace as You Code from Command Line and Export Results"
"Options Files for Polyspace Analysis"
"Baseline Polyspace as You Code Results on Command Line"

# polyspace-bug-finder

(DOS/UNIX) Run a Bug Finder analysis from Windows, Linux, or other command line

## Syntax

```
polyspace-bug-finder [options]
polyspace-bug-finder -sources sourceFiles [options]

polyspace-bug-finder -sources-list-file listOfSources [options]

polyspace-bug-finder -options-file optFile

polyspace-bug-finder -h[elp]
```

## Description

**Note** This Polyspace command is available in *polyspaceroot*\polyspace\bin. Here, *polyspaceroot* is the Polyspace installation folder, for instance, C:\Program Files\Polyspace\R2023a (see also "Installation Folder"). To avoid typing the full path to this command, add this location to the PATH environment variable in your operating system.

polyspace-bug-finder [options] runs a Bug Finder analysis if your current folder contains a sources subfolder with source files (.c or .cxx files). The analysis considers files in sources and all subfolders under sources.

polyspace-bug-finder -sources sourceFiles [options] runs a Bug Finder analysis on the source file(s) sourceFiles. You can customize the analysis with additional options.

polyspace-bug-finder -sources-list-file listOfSources [options] runs a Bug Finder analysis on the source files listed in the text file listOfSources. You can customize the analysis with additional options. Using a sources list file is recommended when you have many source files. By keeping the list of sources in a text file, the command is shorter and updates to the list are easier.

polyspace-bug-finder -options-file optFile runs a Bug Finder analysis with the options specified in the option file. When you have many analysis options, an options file makes it easier to run the same analysis again.

polyspace-bug-finder -h[elp] lists a summary of possible analysis options.

## Examples

### Run Analysis by Directly Specifying Options

Run a local Bug Finder analysis by specifying analysis options at the command line itself. This example uses source files from a demo Polyspace Bug Finder example. To run this example, replace *polyspaceroot* with the path to your Polyspace installation, for example C:\Program Files\Polyspace\R2023a.

Run an analysis on `numerical.c` and `programming.c`, checking for MISRA C:2012 mandatory rules, programming and numerical defects, and using GNU 4.7 compiler settings. This example command is split by ^ characters for readability. In practice, you can put all commands on one line.

```
polyspace-bug-finder -sources ^
polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c,^
polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\programming.c ^
-compiler gnu4.7 -misra3 mandatory -checkers numerical,programming ^
-author jlittle -prog myProject -results-dir C:\Polyspace_Workspace\Results\
```

Open the results.

```
polyspace C:\Polyspace_Workspace\Results\ps_results.psbf
```

To rerun the analysis, you must rerun it from the command line.

### Run Local Analysis with Options File

Run a local Bug Finder analysis by specifying analysis options with an options. This example uses source files from a demo Polyspace Bug Finder example. To run this example, replace *`polyspaceroot`* with the path to your Polyspace installation, for example `C:\Program Files\Polyspace\R2023a`.

Save this text to a text file called `myOptionsFile.txt`.

```
# Options for analyzing numerical.c and programming.c
-sources polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c
-sources polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\programming.c
-compiler gnu4.7
-misra3 mandatory
-checkers numerical,programming
-author jlittle
-prog myProject
-results-dir C:\Polyspace_Workspace\Results\
```

Run the analysis with the options specified in the text file.

```
polyspace-bug-finder -options-file myOptionsFile.txt
```

Open the results.

```
polyspace C:\Polyspace_Workspace\Results\ps_results.psbf
```

To rerun the analysis, you must rerun it from the command line.

## Input Arguments

### `sourceFiles` — Comma-separated names of C or C++ files to analyze

source file name or path

Comma-separated C or C++ source file names, specified as a string. If the files are not in the current folder (pwd), `sourceFiles` must include a full or relative path. To avoid errors because of paths with spaces, add quotes `" "` around the path. For more information, see `-sources`.

If your current folder contains a `sources` subfolder with the source files, you can omit the `-sources` flag. The analysis considers files in `sources` and all subfolders under `sources`.

Example: `myFile.c`, `"C:\mySources\myFile1.c,C:\mySources\myFile2.c"`

**`listOfSources` — Text file listing names of C or C++ files to analyze**
sources list file name or path

Text file which lists the name of C or C++ files, specified as a string. If the files are not in the current folder (`pwd`), `listOfSources` must include a full or relative path. To avoid errors because of paths with spaces, add quotes `" "` around the path. For more information, see `-sources-list-file`.

Example: `filename.txt`, `"C:\ps_analysis\source_files.txt"`

**`options` — Analysis option and corresponding value**
command-line flag with optional value

Analysis options and their corresponding values, specified by the option name and if applicable value. For syntax specifications, see the individual analysis option reference pages.

Example: `-lang C-CPP` `-compiler diab`

**`optFile` — Text file listing analysis options and values**
options file name or path

Text file listing analysis options and values, specified as a string. For more information, see `-options-file`.

Example: `opts.txt`, `"C:\ps_analysis\options.txt"`

## Tips

If you run the command as part of a script, check the exit status to confirm a successful analysis. The command returns zero on a successful analysis. A nonzero return value means that the analysis failed and was not completed. For instance, if the analyzed file does not compile, the command returns a nonzero value. If some of the files do not compile when you are analyzing multiple files, the command completes analysis on the files that do compile and returns zero. It is possible to stop analysis if a file does not compile. See `Stop analysis if a file does not compile (-stop-if-compile-error)`.

After running the command, you can check the `%ERRORLEVEL%` variable in Windows command line to confirm a successful analysis.

# Version History

**Introduced in R2013b**

## See Also

`polyspaceBugFinder`

### Topics

“Run Polyspace Analysis from Command Line”
“Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”
“Complete List of Polyspace Bug Finder Analysis Engine Options”

# polyspace-bug-finder-server

(DOS/UNIX) Run a Bug Finder analysis on a server from Windows, Linux, or other command line

## Syntax

```
polyspace-bug-finder-server [options]
polyspace-bug-finder-server -sources sourceFiles [options]

polyspace-bug-finder-server -sources-list-file listOfSources [options]

polyspace-bug-finder-server -options-file optFile

polyspace-bug-finder-server -h[elp]
```

## Description

**Note** This Polyspace command is available in *polyspaceroot*\polyspace\bin. Here, *polyspaceroot* is the Polyspace Server installation folder, for instance, C:\Program Files \Polyspace Server\R2023a (see also “Installation Folder”). To avoid typing the full path to this command, add this location to the PATH environment variable in your operating system.

polyspace-bug-finder-server [options] runs a Bug Finder analysis on a server if your current folder contains a sources subfolder with source files (.c or .cxx files). The analysis considers files in sources and all subfolders under sources.

polyspace-bug-finder-server -sources sourceFiles [options] runs a Bug Finder analysis on a server on the source file(s) sourceFiles. You can customize the analysis with additional options.

polyspace-bug-finder-server -sources-list-file listOfSources [options] runs a Bug Finder analysis on a server on the source files listed in the text file listOfSources. You can customize the analysis with additional options. Using a sources list file is recommended when you have many source files. By keeping the list of sources in a text file, the command is shorter and updates to the list are easier.

polyspace-bug-finder-server -options-file optFile runs a Bug Finder analysis on a server with the options specified in the option file. When you have many analysis options, an options file makes it easier to run the same analysis again.

polyspace-bug-finder-server -h[elp] lists a summary of possible analysis options.

## Examples

### Run Analysis by Directly Specifying Options

Run a Bug Finder analysis on a server by specifying analysis options in the run command itself. This example uses source files from a demo Polyspace Bug Finder example. To run this example, replace *polyspaceroot* with the path to your Polyspace installation, for example `C:\Program Files\Polyspace\R2023a`.

Run an analysis on `numerical.c` and `programming.c`, checking for MISRA C:2012 mandatory rules, programming and numerical defects, and using GNU 4.7 compiler settings. This example command is split by ^ characters for readability. In practice, you can put all commands on one line.

```
polyspace-bug-finder-server -sources ^
polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c,^
polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\programming.c ^
-compiler gnu4.7 -misra3 mandatory -checkers numerical,programming ^
-author jlittle -prog myProject -results-dir C:\Polyspace_Workspace\Results\
```

After analysis, you can upload the results to the Polyspace Access interface for review. See:

- `polyspace-access`
- “Run Polyspace Bug Finder on Server and Upload Results to Web Interface”

#### Run Analysis with Options File

Run a Bug Finder analysis on a server by specifying analysis options with an options file. This example uses source files from a demo Polyspace Bug Finder example. To run this example, replace *polyspaceroot* with the path to your Polyspace installation, for example `C:\Program Files\Polyspace\R2023a`.

Save this text to a text file called `myOptionsFile.txt`.

```
# Options for analyzing numerical.c and programming.c
-sources polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\numerical.c
-sources polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources\programming.c
-compiler gnu4.7
-misra3 mandatory
-checkers numerical,programming
-author jlittle
-prog myProject
-results-dir C:\Polyspace_Workspace\Results\
```

Run the analysis with the options specified in the text file.

```
polyspace-bug-finder-server -options-file myOptionsFile.txt
```

After analysis, you can upload the results to the Polyspace Access interface for review. See:

- `polyspace-access`
- “Run Polyspace Bug Finder on Server and Upload Results to Web Interface”

## Input Arguments

### `sourceFiles` — Comma-separated names of C or C++ files to analyze

source file name or path

Comma-separated C or C++ source file names, specified as a string. If the files are not in the current folder (pwd), `sourceFiles` must include a full or relative path. To avoid errors because of paths with spaces, add quotes `" "` around the path. For more information, see `-sources`.

If your current folder contains a `sources` subfolder with the source files, you can omit the `-sources` flag. The analysis considers files in `sources` and all subfolders under `sources`.

Example: `myFile.c`, `"C:\mySources\myFile1.c,C:\mySources\myFile2.c"`

**`listOfSources` — Text file listing names of C or C++ files to analyze**
sources list file name or path

Text file which lists the name of C or C++ files, specified as a string. If the files are not in the current folder (`pwd`), `listOfSources` must include a full or relative path. To avoid errors because of paths with spaces, add quotes `" "` around the path. For more information, see `-sources-list-file`.

Example: `filename.txt`, `"C:\ps_analysis\source_files.txt"`

**`options` — Analysis option and corresponding value**
command-line flag with optional value

Analysis options and their corresponding values, specified by the option name and if applicable value. For syntax specifications, see the individual analysis option reference pages.

Example: `-lang C-CPP` `-compiler diab`

**`optFile` — Text file listing analysis options and values**
options file name or path

Text file listing analysis options and values, specified as a string. For more information, see `-options-file`.

Example: `opts.txt`, `"C:\ps_analysis\options.txt"`

## Tips

If you run the command as part of a script, check the exit status to confirm a successful analysis. The command returns zero on a successful analysis. A nonzero return value means that the analysis failed and was not completed. For instance, if the analyzed file does not compile, the command returns a nonzero value. If some of the files do not compile when you are analyzing multiple files, the command completes analysis on the files that do compile and returns zero. It is possible to stop analysis if a file does not compile. See `Stop analysis if a file does not compile (-stop-if-compile-error)`.

After running the command, you can check the `%ERRORLEVEL%` variable in Windows command line to confirm a successful analysis.

# Version History

**Introduced in R2019a**

## See Also

### Topics

“Run Polyspace Bug Finder on Server and Upload Results to Web Interface”
“Complete List of Polyspace Bug Finder Analysis Engine Options”

# polyspace-cluster-profile-manager

(DOS/UNIX) Import MATLAB cluster profile to Polyspace Server commands

## Syntax

```
polyspace-cluster-profile-manager -import-profile clusterProfile
```

## Description

**Note** To avoid typing the full path to this command, add *polyspaceroot*\polyspace\bin to your PATH environment variable . Here, *polyspaceroot* is the Polyspace Server installation folder, for instance, C:\Program Files\Polyspace Server\R2023a.

polyspace-cluster-profile-manager -import-profile clusterProfile imports the cluster profile stored in clusterProfile to your Polyspace Server product. This is equivalent to using the **Cluster Profile Manager** to import a profile. See “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox). Use this command to import a profile if you do not have access to the **Cluster Profile Manager** interface, for instance if you are on a server machine with no graphical interface.

Once you import a profile, you can offload a Polyspace analysis to a remote server by specifying only the name of the job scheduler with option -scheduler.

## Examples

### Import Profile of Cluster that Requires a Certificate

When you enable the flag REQUIRE_CLIENT_CERTIFICATE in file *matlabroot*\toolbox\parallel\bin\mjs_def.sh, clients must use a cluster profile that contains the correct certificate to connect to the job scheduler. Here, *matlabroot* is the installation folder of the MATLAB Parallel Server product, for example C:\Program Files\MATLAB\R2023a.

After you configure the MATLAB Parallel Server cluster and create the certificate file, export the cluster profile and then import it to the Polyspace clients that you want to allow to connect to the job scheduler.

To create a certificate file, generate a shared secret file and then use that file to create the certificate. At the command line, navigate to *matlabroot*\toolbox\parallel\bin\ and run the createSharedSecret script to generate file mySecret.

```
createSharedSecret -file mySecret
```

Pass file mySecret to the generateCertificate script to create the certificate.

```
generateCertificate -secretfile mySecret -certfile clusterClientCert
```

The command generates a certificate file clusterClientCert.

Export the cluster profile by using the createProfile script. Specify the name of the cluster, the hostname of the machine where the job scheduler is running, and the path of the certificate file. For instance, if the job scheduler runs on a machine with hostname mjsHost, run this command:

```
createProfile -name clusterName -host mjsHost -certfile clusterClientCert -outfile mjsProfile
```

The command creates a cluster profile XML file mjsProfile. This XML file contains the certificate that the clients need to connect to the job scheduler.

Import the cluster profile mjsProfile to the client machine where you run the Polyspace Server product that offloads the analysis to the MATLAB Parallel Server cluster. On the machine where you run the Polyspace client, run this command:

```
polyspace-cluster-profile-manager -import-profile path/to/mjsProfile
```

The Polyspace client uses the imported cluster profile and can connect to the job scheduler. Once you import the profile, specify the cluster name with option -scheduler when you run a Polyspace Server analysis in -batch mode, for instance -batch -scheduler clusterName.

## Input Arguments

**clusterProfile — XML file**
source file name or path

clusterName.settings XML file that you generate using the **Cluster Profile Manager** or the createProfile script located in the folder *matlabroot*\toolbox\parallel\bin. Here, *matlabroot* is the installation folder of the MATLAB Parallel Server product, for example C:\Program Files\MATLAB\R2023a.

Example: myClusterName.settings

# Version History

**Introduced in R2021b**

## See Also

Run Bug Finder or Code Prover analysis on a remote cluster (-batch)

### Topics

“Offload Polyspace Analysis from Continuous Integration Server to Another Server”

# polyspace-configure

(DOS/UNIX) Create Polyspace project from your build system at the DOS or UNIX command line

## Syntax

```
polyspace-configure buildCommand
```

```
polyspace-configure [options] buildCommand
```

```
polyspace-configure [options] -compilation-database jsonFile
```

## Description

**Note** This Polyspace command is available in *polyspaceroot*\polyspace\bin. Here, *polyspaceroot* is the Polyspace installation folder, for instance, C:\Program Files\Polyspace\R2023a (see also “Installation Folder” for desktop products or “Installation Folder” for server products). To avoid typing the full path to this command, add this location to the PATH environment variable in your operating system.

polyspace-configure buildCommand traces your build system and creates a Polyspace project with information gathered from your build system. Polyspace creates a project from your build system only if your build commands or makefiles meet certain requirements. For a list of compiler and build command requirements, including a list of compilers that Polyspace supports, see “Requirements for Project Creation from Build Systems”.

polyspace-configure [options] buildCommand traces your build system and uses options to modify the default operation of polyspace-configure. Specify the modifiers before buildCommand, otherwise they are considered as options in the build command itself.

polyspace-configure [options] -compilation-database jsonFile creates a Polyspace project with information gathered from the JSON compilation database file jsonFile that you provide. You do not need to specify a build command or trace your build system. For more on JSON compilation databases, see JSON Compilation Database.

## Examples

### Create Polyspace Project Using Compile Command (gcc, cl, etc.)

This example shows how to use a compile command to create your Polyspace project. The simplest way to run polyspace-configure is to run it directly on the compile command used to build your source code. This example uses the GCC compiler and the compile command gcc. GCC is a standard compiler suite included with Linux based systems. For Windows based systems, MinGW can be installed to support the GCC compiler.

First, ensure that your source code compiles without errors.

```
gcc myFilename.c otherFile.c
```

After confirming that your source code compiles without errors, remove the object files before continuing.

```
rm myFilename.o otherFile.o
```

Create a Polyspace project specifying a unique project name. Run `polyspace-configure` on the compile command as previously tested.

```
polyspace-configure -prog myProject gcc myFilename.c otherFile.c
```

Open the Polyspace project in the Polyspace user interface.

If you use Visual Studio to compile your sources, you can replace `gcc` in the above example with `cl`. You can also run `polyspace-configure` on a full build of your Visual Studio project. See "Create Polyspace Projects from Visual Studio Build" .

Note that `polyspace-configure` creates a new Polyspace project using the sources from your compile command. It is not possible to append to an already existing Polyspace project by using the `polyspace-configure` command. If you use compile options (such as `-D` for `gcc`), `polyspace-configure` sets the equivalent Polyspace options in the project. For more details, see "Algorithms" on page 4-49.

### Create Polyspace Project from Makefile

Typically, in real applications, you do not invoke the compiler directly to build your source code. Your build command runs several commands, one of which is compilation. For instance, you might be running a makefile to build your source code. This example shows how to create a Polyspace project if you use the command `make` *`targetName buildOptions`* to build your source code.

Create a Polyspace project specifying a unique project name. Use the `-B` or `-W` *`makefileName`* option with `make` so that the all prerequisite targets in the makefile are remade.

```
polyspace-configure  -prog myProject \
make -B targetName buildOptions
```

Open the Polyspace project in the Polyspace user interface.

### Create Polyspace Options File from JSON Compilation Database

This example shows how to create a Polyspace options file from a JSON compilation database that you generate with the CMake build system generator. CMake generates build instructions for the build tool you specify, such as a Unix Makefiles for `make` or project files for Microsoft Visual Studio. CMake supports the generation of a JSON compilation database only for Makefile generators and Ninja generator. For more information, see makefile generators.

Generate a JSON compilation database for your CMake project. For an example of a Cmake project, see *`polyspaceroot`*`\help\toolbox\bugfinder\examples\compilation_database` where *`polyspaceroot`* is your Polyspace installation folder.

Navigate to the root of your project source tree. This folder contains the file `CMakeLists.txt` which CMake uses as an input to generate build instructions. Enter these commands:

```
mkdir JSON_cdb
cd JSON_cdb
cmake -G "Unix Makefiles" -DCMAKE_EXPORT_COMPILE_COMMANDS=1 ../
```

The last command generates a Unix makefile with build instructions for the `make` build tool. The command also outputs file `compile_commands.json`. This file lists the compiler calls for every translation unit in your project.

Generate a Polyspace options file from the compilation database that you generated in the previous step.

```
polyspace-configure -compilation-database compile_commands.json \
-output-options-file options.txt
```

You do not need to specify a build command and `polyspace-configure` does not trace your build. Polyspace extracts information about your build system from the JSON compilation database.

Pass the options file to Polyspace to run an analysis, for instance:

```
polyspace-bug-finder -options-file options.txt
```

### Create Projects That Have Different Source Files from Same Build Trace

This example shows how to create different Polyspace projects from the same trace of your build system. You can specify which source files to include for each project.

Trace your build system without creating a Polyspace project by specifying the option `-no-project`. To ensure that all the prerequisite targets in your makefile are remade, use the appropriate `make` build command option, for instance `-B`.

```
polyspace-configure -no-project make -B
```

`polyspace-configure` stores the cache information and the build trace in default locations inside the current folder. To store the cache information and build trace in a different location, specify the options `-cache-path` and `-build-trace`.

Generate Polyspace projects by using the build trace information from the previous step. Specify a project name and use the `-include-sources` or `-exclude-sources` option to select which files to include for each project.

```
polyspace-configure -no-build -prog myProject \
-include-sources "glob_pattern"
```

*`glob_pattern`* is a glob pattern that corresponds to folders or files you filter in or out of your project. To ensure the shell does not expand the glob patterns you pass to `polyspace-configure`, enclose them in double quotes. For more information on the supported syntax for glob patterns, see “Select Files for Polyspace Analysis Using Pattern Matching”.

If you specified the options `-build-trace` and `-cache-path` in the previous step, specify them again.

Delete the trace file and cache folder.

```
rm -r polyspace_configure_cache polyspace_configure_built_trace
```

If you used the options `-build-trace` and `-cache-path`, use the paths and file names from those options.

### Run Command-Line Polyspace Analysis from Makefile

This example shows how to run Polyspace analysis if you use the command `make` *`targetName buildOptions`* to build your source code. In this example, you use `polyspace-configure` to trace your build system but do not create a Polyspace project. Instead you create an options file that you can use to run Polyspace analysis from command-line.

Create a Polyspace options file specifying the `-output-options-file` command. Use the `-B` or `-W` *`makefileName`* option with `make` so that all prerequisite targets in the makefile are remade.

```
polyspace-configure -output-options-file\
 myOptions make -B targetName buildOptions
```

Use the options file that you created to run a Polyspace analysis at the command line:

```
polyspace-bug-finder -options-file myOptions
```

## Input Arguments

### `buildCommand` — Command for full build of source code

string

Build command specified exactly as you use to build your source code.

The build command must perform a full build of your sources and not an incremental build. Typically, build commands such as `make` are set up to only build sources that have changed since the previous build. However, `polyspace-configure` requires a full build to determine which sources to add to a Polyspace project or options file. Add appropriate options to your build command to force a full build. For instance, when using a makefile, append the option `-B` to `make`.

Example: `make -B`, `make -B -W` *`makefileName`*

### `options` — Options for changing default operation of `polyspace-configure`

single option starting with `-`, followed by argument | multiple space-separated option-argument pairs

**Basic Options**

| Option | Argument | Description |
|---|---|---|
| `-prog` | Project name | Project name that appears in the Polyspace user interface. The default is `polyspace`.<br>If you do not use the option `-output-project`, the `-prog` argument also sets the project name.<br>**Example:** `-prog myProject` creates a project that has the name `myProject` in the user interface. If you do not use the option `-output-project`, the project name is also `myProject.psrprj`. |

| Option | Argument | Description |
|---|---|---|
| -author | Author name | Name of project author.<br>**Example:** -author jsmith |
| -output-project | Path | Name and location of Polyspace project. The default is the file polyspace.psprj in the current folder.<br>**Example:** -output-project ../myProjects/project1 creates a project project1.psprj in the folder with the relative path ../myProjects/. |
| -output-options-file | File name | Option to create a Polyspace analysis options file. Use this file for command-line analysis using one of these commands:<br>• polyspace-bug-finder<br>• polyspace-code-prover<br>• polyspace-bug-finder-server<br>• polyspace-code-prover-server<br>• polyspace-bug-finder-access |
| -allow-build-error | None | Option to create a Polyspace project even if an error occurs in the build process.<br>If an error occurs, the build trace log shows the following message:<br>polyspace-configure (polyspaceConfigure)<br>   ERROR: build command<br>   *command_name* fail [status=*status_value*]<br>*command_name* is the build command name that you use and *status_value* is the non-zero exit status or error level that indicates which error occurred in your build process.<br>This option is ignored when you use -compilation-database. |
| -allow-overwrite | None | Option to overwrite a project with the same name, if it exists.<br>By default, polyspace-configure (polyspaceConfigure) throws an error if a project with the same name already exists in the output folder. Use this option to overwrite the project. |
| -no-console-output<br>-silent (default)<br>-verbose | None | Option to suppress or display additional messages from running polyspace-configure (polyspaceConfigure).<br>• -no-console-output – Suppress all outputs including errors and warnings.<br>• -silent (default) – Show only errors and warnings.<br>• -verbose – Show all messages.<br>If you specify more than one of these options, the most verbose option is applied.<br>These options are ignored if they are used in combination with -easy-debug. |

| Option | Argument | Description |
|---|---|---|
| `-help` | None | Option to display the full list of `polyspace-configure (polyspaceConfigure)` commands |
| `-debug` | None | Option to store debug information for use by MathWorks technical support.<br>This option has been superseded by the option `-easy-debug`. |
| `-easy-debug` | Path | Option to store debug information for use by MathWorks technical support.<br>After a `polyspace-configure (polyspaceConfigure)` run, the path provided contains a zipped file ending with `pscfg-output.zip`. If the run fails to create a complete Polyspace project or options file, send this zipped file to MathWorks Technical Support for further debugging. The zipped file does not contain source files traced in the build. See also “Errors in Project Creation from Build Systems”. |

**Options to Create Multiple Modules**

These options are not compatible with `-compilation-database`.

| Option | Argument | Description |
|---|---|---|
| `-module` | None | Option to create a separate options file for each binary you create in build system. You can only create separate options files for different binaries.<br>Use this option only if your build system uses GNU or Visual C++ compilers.<br>See also “Modularize Polyspace Analysis by Using Build Command”. |
| `-output-options-path` | Path name | Location where generated options files are saved. Use this option together with the option `-module`.<br>The options files are named after the binaries created in the build system. |

**Advanced Options**

| Option | Argument | Description |
|---|---|---|
| -compilation-database | Path and file name | Location and name of JSON compilation database (JSON CDB) file. You generate this file from your build system, for instance by using the flag -DCMAKE_EXPORT_COMPILE_COMMANDS=1 with cmake. The file contains compiler calls for all the translation units in you projects. For more information, see JSON Compilation Database. polyspace-configure uses the content of this file to get information about your build system. The extracted compiler paths in the JSON CDB must be accessible from the path where you run polyspace-configure.<br>You do not specify a build command when you use this option.<br>These build systems and compilers support the generation of a JSON CDB:<br>• CMake<br>• Bazel<br>• Clang<br>• Ninja<br>• Qbs<br>• waf<br>This option is not compatible with -no-project and with the options to create multiple modules.<br>The cache control options, -allow-build-error, and -no-build are ignored when you use this option. |
| -compiler-config | Path and file name | Location and name of compiler configuration file.<br>The file must be in a specific format. For guidance, see the existing configuration files in *polyspaceroot*\polyspace\configure\compiler_configuration\. For information on the contents of the file, see “Create Polyspace Projects from Build Systems That Use Unsupported Compilers”.<br>**Example:** -compiler-configuration myCompiler.xml |
| -no-project | None | Option to trace your build system without creating a Polyspace project and save the build trace information.<br>Use this option to save your build trace information for a later run of polyspace-configure (polyspaceConfigure) with the -no-build option.<br>This option is not compatible with -compilation-database. |

| Option | Argument | Description |
|---|---|---|
| `-no-build` | None | Option to create a Polyspace project using previously saved build trace information.<br>To use this option, you must have the build trace information saved from an earlier run of `polyspace-configure (polyspaceConfigure)` with the `-no-project` option.<br>If you use this option, you do not need to specify the `buildCommand` argument.<br>This option is ignored when you use `-compilation-database`. |
| `-no-sources` | None | Option to create a Polyspace options file that does not contain the source file specifications.<br>Use this option when you intend to specify the source files by other means. For instance, you can use this option when:<br>• Running Polyspace on AUTOSAR-specific code.<br>You want to create an options file that traces your build command for the compiler options:<br>`-output-options-file options.txt -no-sources`<br>You later append this options file when extracting source file names from ARXML specifications and running the subsequent Code Prover analysis with `polyspace-autosar`<br>`-extra-options-file options.txt`<br>See also “Run Polyspace on AUTOSAR Code Using Build Command” (Polyspace Code Prover).<br>• Running Polyspace in Eclipse™.<br>Your source files are already specified in your Eclipse project. When running a Polyspace analysis, you want to specify an options file that has the compilation options only. |

| Option | Argument | Description |
|---|---|---|
| `-extra-project-options` | Options to use for subsequent Polyspace analysis. For instance, `"-stubbed-pointers-are-unsafe"`. | Options that are used for subsequent Polyspace analysis.<br>Once a Polyspace project is created, you can change some of the default options in the project. Alternatively, you can pass these options when tracing your build command. The flag `-extra-project-options` allows you to pass additional options.<br>Specify multiple options in a space separated list, for instance `"-allow-negative-operand-in-shift -stubbed-pointers-are-unsafe"`.<br>Suppose you have to set the option `-stubbed-pointers-are-unsafe` for every Polyspace project created. Instead of opening each project and setting the option, you can use this flag when creating the Polyspace project:<br>`-extra-project-options "-stubbed-pointers-are-unsafe"`<br>For the list of options available, see:<br>• "Complete List of Polyspace Bug Finder Analysis Engine Options"<br>• "Complete List of Polyspace Code Prover Analysis Options" (Polyspace Code Prover)<br>If you are creating an options file instead of a Polyspace project from your build command, do not use this flag. Instead, add the extra analysis options manually in the generated options file, in a separate options file, or at the command-line when you start the analysis. |
| `-tmp-path` | Path | Location of folder where temporary files are stored. |
| `-build-trace` | Path and file name | Location and name of file where build information is stored. The default is `./polyspace_configure_build_trace.log`.<br>**Example:** `-build-trace ../build_info/trace.log` |
| `-log` | Path and file name | Location and name of log file where the output of the `polyspace-configure` command is stored. The use of this option does not suppress the console output. |
| `-include-sources`<br>`-exclude-sources` | Glob pattern | Option to specify which source files `polyspace-configure (polyspaceConfigure)` includes in, or excludes from, the generated project. You can combine both options together.<br>A source file is included if the file path matches the glob pattern that you pass to `-include-sources`.<br>A source file is excluded if the file path matches the glob pattern that you pass to `-exclude-sources`. |

| Option | Argument | Description |
|---|---|---|
| -print-included-sources<br>-print-excluded-sources | None | Option to print the list of source files that polyspace-configure (polyspaceConfigure) includes in, or excludes from, the generated project. You can combine both options together. The output displays the full path of each file on a separate line.<br>Use this option to troubleshoot the glob patterns that you pass to -include-sources or -exclude-sources. You can see which files match the pattern that you pass to -include-sources or -exclude-sources. |
| -compiler-cache-path | Folder path | Specify a folder path where polyspace-configure looks for or stores the compiler cache files. If the folder does not exist, polyspace-configure creates it.<br>By default, Polyspace looks for and stores compiler caches under these folder paths:<br>• **Windows**<br>%appdata%\Mathworks\R20xxY\Polyspace<br>• **Linux**<br>~/.matlab/R20xxY/Polyspace<br>• **Mac**<br>~/Library/Application Support/MathWorks/MATLAB/R20xxY/Polyspace<br>*R20xxY* is the release version of your Polyspace product, for instance R2020b. |
| -no-compiler-cache | None | Use this option if you do not want Polyspace to cache your compiler configuration information or to use an existing cache for your compiler configuration.<br>By default, the first time you run polyspace-configure with a particular compiler configuration, Polyspace queries your compiler for the size of fundamental types, compiler macro definitions, and other compiler configuration information then caches this information. Polyspace reuses the cached information in subsequent runs of polyspace-configure for builds that use the same compiler configuration. |
| -reset-compiler-cache-entry | None | Use this option to query the compiler for the current configuration and to refresh the entry in the cache file that corresponds to this configuration. Other compiler configuration entries in the cache are not updated. |
| -clear-compiler-cache | None | Use this option to delete all compiler configurations stored in the cache file.<br>If you also specify a build command or -compilation-database, polyspace-configure computes and caches the compiler configuration information of the current run, except if you specify -no-project or -no-compiler-cache. |

| Option | Argument | Description |
|---|---|---|
| `-import-macro-definitions` | `none`<br>`from-allowlist`<br>`from-source-tokens`<br>`from-compiler` | Typically, you do not need to specify this option.<br>Polyspace attempts to automatically determine the best strategy to query your compiler for macro definitions in this order of priority:<br>1 `from-compiler` — Polyspace uses native compiler options, such as `gcc -dm -E`, to obtain the compiler macro definitions. This strategy does not require Polyspace to trace your build and is available only for compilers that support listing macro definitions.<br>2 `from-source-tokens` — Polyspace uses every non-keyword token in your source code to query your compiler for macro definitions. This strategy is available only if Polyspace can trace your build. The strategy is not available if you use option `-compilation-database`.<br>3 `from-allowlist` — Polyspace uses an internal allow list to query the compiler for macro definitions.<br>If you prefer to specify macro definitions manually, use this option with the `none` flag and use option `Preprocessor definitions (-D)` to specify the macro definitions.<br>If the macro import strategy that Polyspace uses is not the one that you expect, try specifying this option manually to troubleshoot the issue. |
| `-options-for-sources-delimiter` | A single character | Specify an option separator to use when multiple analysis options are associated with one source file using the `-options-for-sources` option. Typically, the `-options-for-sources` option uses a semicolon as separator.<br>See also `-options-for-sources`. |

### Cache Control Options

These options are primarily useful for debugging. Use the options if `polyspace-configure (polyspaceConfigure)` fails and MathWorks Technical Support asks you to use the option and provide the cached files. Starting R2020a, the option `-easy-debug` provides an easier way to provide debug information. See “Contact Technical Support About Issues with Running Polyspace”.

These options are ignored when you use `-compilation-database`.

| Option | Argument | Description |
|---|---|---|
| `-no-cache`<br>`-cache-sources` (default)<br>`-cache-all-text`<br>`-cache-all-files` | None | Option to perform one of the following:<br>• `-no-cache`: Not create a cache<br>• `-cache-sources`: Cache text files temporarily created during build for later use by `polyspace-configure (polyspaceConfigure)`.<br>• `-cache-all-text`: Cache all text files including sources and headers.<br>• `-cache-all-files`: Cache all files including binaries.<br>Typically, you cache temporary files created by your build command to debug issues in tracing the command. |

| Option | Argument | Description |
|---|---|---|
| `-cache-path` | Path | Location of folder where cache information is stored.<br>When tracing a Visual Studio build (`devenv.exe`), if you see the error:<br>`path is too long`<br>try using a shorter path for this option to work around the error.<br>**Example:** `-cache-path ../cache` |
| `-keep-cache`<br>`-no-keep-cache` (default) | None | Option to preserve or clean up cache information after `polyspace-configure (polyspaceConfigure)` completes execution.<br>If `polyspace-configure (polyspaceConfigure)` fails, you can provide this cache information to technical support for debugging purposes. |

## Algorithms

The `polyspace-configure` command creates a Polyspace project or options file from a build command such as `make` using roughly these steps:

1. The build command is first executed. `polyspace-configure` keeps track of the commands that run during this build and the files read or written. One or more of these commands would be the compiler invocation. For instance, if the build command uses a GCC compiler, one or more of these commands would exercise the `gcc`, `g++`, or related executable. Based on the presence of a known compiler executable, `polyspace-configure` picks out the compiler invocation commands from amidst all the commands executed during build.
2. Each compiler invocation command contains these three parts: the compiler executable, some source files and some compiler options. For instance, the following command exercises the GCC compiler on the file `myFile.c` with a compiler option `-std` that triggers C++11-based compilation:

   ```
   gcc -std=c++11 myFile.c
   ```

   `polyspace-configure` reads the source file names from these commands and directly uses them in the Polyspace project or options file. The compiler executable plus compiler options are translated to Polyspace analysis options.

   To determine Polyspace options such as the ones corresponding to sizes of basic types or underlying type of `size_t`, `polyspace-configure` runs the previously read compiler executable plus compiler options on some small source files. Depending on whether a source file compiles successfully or shows errors, `polyspace-configure` can set an appropriate Polyspace option. To determine compiler macro definitions and include paths, `polyspace-configure` also reinvokes the compiler on small sources but thereafter uses a slightly different strategy.

   For a simple example of a source file that can help determine a Polyspace option, see the reference page for the option `Management of size_t (-size-t-type-is)`.

Instead of a build command, `polyspace-configure` can also create a project or options file from a JSON compilation database. When the `polyspace-configure` command runs on a compilation database, the first step above is omitted. A compilation database directly states the compiler invocation command in entries such as this:

```
{
"directory": "/proj/files/
"command": "/usr/local/bin/gcc -std=c++11 -c /proj/files/myFile.c",
"file" : "/proj/files/myFile.c"
}
```

`polyspace-configure` can simply read these compiler invocation commands and continue with the remaining step of reinvoking the compiler on small source files. Since the build command execution step is skipped, running `polyspace-configure` on a compilation database is much faster than running `polyspace-configure` on a build command. However, it is your responsibility to make sure that the compilation database you provide accurately reflects a full build of your source code.

# Version History

**Introduced in R2013b**

## See Also

### Topics

“Requirements for Project Creation from Build Systems”
“Create Polyspace Projects from Build Systems That Use Unsupported Compilers”
“Create Polyspace Analysis Configuration from Build Command (Makefile)”
“Modularize Polyspace Analysis by Using Build Command”

# polyspace-report-generator

(DOS/UNIX) Generate reports for Polyspace analysis results stored locally or on Polyspace Access

## Syntax

```
polyspace-report-generator -template outputTemplate [reporting options]
polyspace-report-generator -generate-results-list-file [-results-dir
resultsFolder] [-set-language-english]
polyspace-report-generator -generate-variable-access-file [-results-dir
resultsFolder] [-set-language-english]
polyspace-report-generator -configure-keystore
```

```
polyspace-report-generator -template outputTemplate -host hostName -run-id
runID [polyspace access options] [reporting options]
polyspace-report-generator -generate-results-list-file -host hostName -run-id
runID [polyspace access options] [-set-language-english]
polyspace-report-generator -generate-variable-access-file -host hostName -
run-id runID [polyspace access options] [-set-language-english]
```

## Description

### Generate Reports from Local Results

**Note** This Polyspace command is available in `*polyspaceroot*\polyspace\bin`. Here, `*polyspaceroot*` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a` (see also “Installation Folder” for desktop products or “Installation Folder” for server products). To avoid typing the full path to this command, add this location to the `PATH` environment variable in your operating system.

`polyspace-report-generator -template outputTemplate [reporting options]` generates a report by using the template `outputTemplate` for the local analysis results that you specify with `reporting options`.

By default, reports for results from `project-name` are stored as `project-name_report-name` in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the results folder of `project-name`.

`polyspace-report-generator -generate-results-list-file [-results-dir resultsFolder] [-set-language-english]` exports the analysis results stored locally in `resultsFolder` to a tab-delimited text file. The file contains the result information available on the **Results List** pane in the user interface. For more information on the exported results list, see “View Exported Results”.

By default, the results file for results from `project-name` is stored in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the results folder of `project-name`.

For exporting results to a tab-delimited text file, the `polyspace-results-export` command is preferred.

`polyspace-report-generator -generate-variable-access-file [-results-dir resultsFolder] [-set-language-english]` exports the list of global variables in your code from the Code Prover analysis stored locally in `FOLDER` to a tab-delimited text file. The file contains the information available on the **Variable Access** pane in the user interface. For more information on the exported variables list, see “View Exported Variable List” (Polyspace Code Prover).

By default, the variables file for results from `project-name` is stored in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the results folder of `project-name`.

`polyspace-report-generator -configure-keystore` configures the report generator to communicate with Polyspace Access over HTTPS.

Run this one-time configuration step if Polyspace Access is configured to use the HTTPS protocol and you do not have a Polyspace Bug Finder desktop license, or you have a desktop license but you have not configured the desktop UI to communicate with Polyspace Access over HTTPS. Before running this command, generate a client keystore to store the SSL certificate that Polyspace Access uses for HTTPS. See “Generate a Client Keystore”.

#### Generate Reports from Polyspace Access Results

`polyspace-report-generator -template outputTemplate -host hostName -run-id runID [polyspace access options] [reporting options]` generates a report by using the template `outputTemplate` for the analysis results corresponding to run `runID` on Polyspace Access. `hostName` is the fully qualified host name of the machine that hosts Polyspace Access.

By default, reports for results from `project-name` are stored as `project-name_report-name` in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the path from which you call the command.

`polyspace-report-generator -generate-results-list-file -host hostName -run-id runID [polyspace access options] [-set-language-english]` exports the analysis results corresponding to run `runID` on Polyspace Access to a tab-delimited text file. The file contains the result information available on the **Results List** pane in the Polyspace Access web interface. `hostName` is the fully qualified host name of the machine that hosts Polyspace Access. For more information on the exported results list, see “Results List in Polyspace Access Web Interface”.

By default, the results file for results from `project-name` is stored in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the path from which you call the command.

For exporting results to a tab-delimited text file, the `polyspace-results-export` command is preferred.

`polyspace-report-generator -generate-variable-access-file -host hostName -run-id runID [polyspace access options] [-set-language-english]` exports the list of global variables in your code from the Code Prover analysis corresponding to run `runID` on Polyspace Access to a tab-delimited text file. The file contains the information available on the **Variable Access** pane in the Polyspace Access web interface. `hostName` is the fully qualified host name of the machine that hosts Polyspace Access. For more information on the exported variables list, see “View Exported Variable List” (Polyspace Code Prover).

By default, the variables file for results from `project-name` is stored in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the path from which you call the command.

## Examples

### Generate PDF Reports for Analysis Results Stored Locally

You can generate multiple reports for analysis results that you store locally.

Create a variable `template_path` to store the path to the report templates and create a variable `report_templates` to store a comma-separated list of templates to use.

```
SET template_path="C:\Program Files"\Polyspace\R2019a\toolbox\polyspace^
\psrptgen\templates\bug_finder
SET report_templates=%template_path%\BugFinder.rpt,^
%template_path%\CodingStandards.rpt
```

Generate the reports from the templates that you specified in `report_templates` for analysis results of Polyspace project `myProject`.

```
 polyspace-report-generator -template %report_templates% ^
-results-dir C:\Polyspace_Workspace\myProject\Module_1\BF_Result ^
-format PDF
```

The command generates two PDF reports, `myProject_BugFinder.PDF` and `myProject_CodingStandards.PDF`. The reports are stored in `C:\Polyspace_Workspace\myProject\Module_1\BF_Result\Polyspace-Doc`. For more information on the content of the reports, see `Bug Finder and Code Prover report (-report-template)`.

### Configure Report Generator with Client Keystore

If you configure Polyspace Access to use the HTTPS protocol, you must generate a client keystore where you store the SSL certificate that Polyspace Access uses, and configure `polyspace-report-generator` to use that keystore. See “Generate a Client Keystore”. This one-time configuration enables the report generator to communicate with Polyspace Access over HTTPS.

To configure the report generator with a client keystore, use the `polyspace-report-generator -configure-keystore` command. Follow the prompts to provide the URL you use to log into Polyspace Access, the full path to the keystore file you generated, and the keystore password.

```
polyspace-report-generator -configure-keystore
Location: US, user name: jsmit, id: 62600@us-jsmith, print mode: false
Enter the Polyspace Access URL using form  http[s]://<host>:<port> :
https://myAccessServer:9443
Enter full path to client keystore file :
C:\R2019b\ssl\client-cert.jks
Enter client keystore password :

The keystore has been configured
```

You must run the keystore configuration command again if:

- The Polyspace Access URL changes, for instance if you use a different port number.
- The path to the keystore file changes.
- The keystore password changes.

### Generate Report and Variables List from Polyspace Access

**Note** To generate reports of results on Polyspace Access at the command line, you must have a Polyspace Bug Finder Server or Polyspace Code Prover Server installation.

Suppose that you want to generate a report and export the variables list for the results of a Code Prover analysis stored on the Polyspace Access database.

To connect to Polyspace Access, provide a host name and your login credentials including your encrypted password. To encrypt your password, use the `polyspace-access` command and enter your user name and password at the prompt.

```
polyspace-access -encrypt-password
login: jsmith
password:
CRYPTED_PASSWORD LAMMMEACDMKEFELKMNDCONEAPECEEKPL
Command Completed
```

Store the login and encrypted password in a credentials file and restrict read and write permission on this file. Open a text editor, copy these two lines in the editor, then save the file as `myCredentials.txt` for example.

```
 -login jsmith
 -encrypted-password LAMMMEACDMKEFELKMNDCONEAPECEEKPL
```

To restrict the file permissions, right-click the file and select the **Permissions** tab on Linux or the **Security** tab on Windows.

To specify project results on Polyspace Access, specify the run ID of the project. To obtain a list of projects with their latest run IDs, use the `polyspace-access` with option `-list-project`.

```
polyspace-access -host myAccessServer -credentials-file myCredentials.txt -list-project
Connecting to https://myAccessServer:9443
Connecting as jsmith
Get project list with the last Run Id
Restricted/Code_Prover_Example (Code Prover) RUN_ID 14
public/Bug_Finder_Example (Bug Finder) RUN_ID 24
public/CP/Code_Prover_Example (Polyspace Code Prover) RUN_ID 16
public/Polyspace (Code Prover) RUN_ID 28
Command Completed
```

For more information on this command, see `polyspace-access`.

Generate a `Developer` report for results with run ID `16` from the Polyspace Access instance with host name `myAccessServer`. The URL of this instance of Polyspace Access is `https://myAccessServer:9443`.

```
SET template_path=^
"C:\Program Files\Polyspace\R2019a\toolbox\polyspace\psrptgen\templates"

polyspace-report-generator -credentials-file myCredentials.txt ^
-template %template_path%\Developer.rpt ^
-host myAccessServer ^
-run-id 16 ^
-output-name myReport
```

The command creates report `myReport.docx` by using the template that you specify. The report is stored in folder `Polyspace-Doc` on the path from which you called the command.

Generate a tab-delimited text file that contains a list of global variables in your code for the specified analysis results.

```
polyspace-report-generator -credentials-file myCredentials.txt^
-generate-variable-access-file ^
-host myAccessServer ^
-run-id 16
```

The list of global variables `Variable_View.txt` is stored in the same folder as the generated report. For more information on the exported variables list, see “View Exported Variable List” (Polyspace Code Prover).

## Input Arguments

### `outputTemplate` — path to report template file

string

Path to the report template that you use to generate an analysis report. To generate multiple reports, specify a comma-separated list of report template paths (do not put a space after the commas). The templates are available in *`polyspaceroot`*`\toolbox\polyspace\psrptgen\templates\` as `.rpt` files. Here, *`polyspaceroot`* is the Polyspace installation folder. The `polyspace-report-generator` command accepts the same templates as the analysis option `-report-template`. For information on the available templates, see `Bug Finder and Code Prover report (-report-template)`.

This option is not compatible with `-generate-variable-access-file` and `-generate-results-list-file`.

Example: `C:\Program Files\Polyspace\R2019a\toolbox\polyspace\psrptgen\templates\Developer.rpt`

Example: `TEMPLATE_PATH\BugFinder.rpt,TEMPLATE_PATH\CodingStandards.rpt`

### `resultsFolder` — Analysis results folder path

string

Path to a folder containing Polyspace analysis results (`.psbf` or `.pscp` file). To generate a report for multiple verifications, specify a comma-separated list of folder paths (do not put a space after the commas). If you do not specify a folder path, the command generates a report for analysis results in the current folder.

Example: `C:\Polyspace_Workspace\My_project\Module_1\results`

Example: `C:\Polyspace_Workspace\My_project\Module_2\results,C:\Polyspace_Workspace\My_project\Module_3\other_results`

### `hostName` — Polyspace Access machine host name

string

Fully qualified host name of the machine that hosts the Polyspace Access **Gateway API** service. You must specify a host name to generate a report for results on the Polyspace Access database.

Example: `my-company-server`

### `runID` — Polyspace Access run ID

integer

Run ID of the project findings for which you generate a report. Polyspace assigns a unique run ID to each analysis run that you upload to the Polyspace Access.

You can see the run ID of a project in the Polyspace Access web interface. To get the run ID, use the command `polyspace-access` with option `-list-project`.

### `reporting options` — Options for generated report

string

| Option | Description |
|---|---|
| `-format` *`HTML`* `\|` *`PDF`* `\|` *`WORD`* | File format of the report that you generate. By default, the command generates a WORD document.<br>To generate reports in multiple formats, specify a comma-separated list of formats. (Do not put a space after the commas). For instance, `-format PDF,HTML`.<br>This option is not compatible with `-generate-variable-access-file` and `-generate-results-list-file`. |
| `-output-name` *`outputName`* | Name of the generated report or folder name if you generate multiple reports.<br>The report or exported file is saved on the path from which you call the command. To save in a different folder, specify the full path to the folder, for instance `-output-name C:\PathTo\OtherFolder`. |
| `-results-dir` *`FOLDER_1,...,FOLDER_N`* | Path to the locally stored results folder. To generate reports for multiple analyses, specify a comma-separated list of folder path. (Do not put a space after the commas). For example:<br>`-results-dir folderPath1,folderPath2`<br>This option is not compatible with Polyspace Access results. |
| `-key-mode` *`file-scope`* `\|` *`function-scope`* | Specify how the entries in the **Key** column of the exported results are calculated:<br>• `file-scope` (default) — The key is calculated from the result name, result type, and file location.<br>• `function-scope` — The key is calculated from the result name, result type, and function location if the results is located inside a function, or file location otherwise. Keys generated in this mode are prefixed with `FN` if the result is inside a function.<br>See also “Enable Function Scope for Exported Keys”.<br>This option applies only in combination with option `-generate-results-list-file`. |

| Option | Description |
|---|---|
| `-set-language-english` | Generate the report in English. Use this option if your display language is set to another language. |
| `-help` | Display the help information. |

**`polyspace access options` — Options for Polyspace Access**
string

| Option | Description |
|---|---|
| `-host` *`HOST_NAME`* | `HOST_NAME` corresponds to the host name that you specify in the URL of the Polyspace Access interface, for example `https://`*`HOST_NAME`*`:`*`port`*`/metrics/index.html`. If you are unsure about which host name to use, contact your Polyspace Access administrator. The default host name is `localhost`.<br>This option is mandatory when you generate reports for results stored on the Polyspace Access database. |
| `-run-id` *`RUN_ID`* | Run ID of the project. Polyspace assigns a unique run ID to each analysis run that you upload. To get the last run ID of a project, use the `-list-project` option of the `polyspace-access` command.<br>For more information on the command, see `polyspace-access`.<br>This option is mandatory when you generate reports for results stored on the Polyspace Access database. |
| `-all-units` | Specify this option to generate a report for all units from a `unit by unit` analysis.<br>When you use this option, specify the run ID of only one unit with `-run-id`. The command includes the other units from the analysis in the report. |
| `-port` *`portNumber`* | `portNumber` corresponds to the port number that you specify in the URL of the Polyspace Access interface, for example `https://`*`hostname`*`:`*`portNumber`*`/metrics/index.html`. If you are unsure about which port number to use, contact your Polyspace Access administrator. The default port number is `9443`. |
| `-protocol` *`http`* `\|` *`https`* | HTTP protocol used to connect to Polyspace Access. Default value is `https`. |

| Option | Description |
|---|---|
| `-credentials-file` *`file_path`* | Full path to the text file where you store your login credentials. Use this option if, for instance, you use a command that requires your Polyspace Access credentials in a script but you do not want to store your credentials in that script. While the script runs, someone inspecting currently running processes cannot see your credentials.<br>You can store only one set of credentials in the file, either as `-login` and `-encrypted-password` entries on separate lines, for instance:<br>`-login jsmith`<br>`-encrypted-password LAMMMEACDMKEFELKMNDCONEAPECEEKPL`<br>or as an `-api-key` entry:<br>`-api-key keyValue123`<br>Make sure that you restrict the read and write permissions on the file where you store your credentials. For example, to restrict read and write permissions on file `login.txt` in Linux, use this command:<br>`chmod go-rwx login.txt` |
| `-api-key` *`keyValue`* | API key you use as a login credential instead of providing your login and encrypted password. To assign an API key to a user, see “Configure User Manager” or contact your Polyspace Access administrator.<br>Use the API key if, for instance, you use a command that requires your Polyspace Access login credentials as part of an automation script with a CI tool like Jenkins. If a user updates his or her password, you do not need to update the API key associated with that user in your scripts.<br>It is recommended that you store the API key in a text file and pass that file to the command by using `-credentials-file`. See the description for option `-credentials-file`. |
| `-login` *`username`*<br>`-encryted-password` *`ENCRYPTED_PASSWD`* | Credentials that you use to log into Polyspace Access. The argument of `-encrypted-password` is the output of the `polyspace-access -encrypt-password` command.<br>For more information on the command, see `polyspace-access`. |

## Tips

- You cannot use the `polyspace-report-generator` command with results generated with Polyspace as You Code. Use the `polyspace-results-export` command instead. See `polyspace-results-export`.
- Report generation can take a long time for result sets containing a very large number of defects or coding rules violations. In some cases, you might run into issues from insufficient memory. If that happens, you can either increase the Java® heap size or use the option `-wysiwyg` to generate multiple filtered reports instead of a single report containing all results. For more information, see “Fix Insufficient Memory Errors During Polyspace Report Generation”.

## Alternative Functionality

Instead of generating reports from existing analysis results, you might want a report to be generated along with the analysis. In this case, use the analysis options `-report-template` and `-report-output-format`. See “Reporting”.

Reports generated along with analysis will only contain review information imported from previous analyses (if any such information is imported). If you add new review information to the results, you will have to regenerate the reports using the `polyspace-report-generator` command.

# Version History

**Introduced in R2013b**

## See Also

`polyspace-results-export`

### Topics

“Generate Reports from Polyspace Results”
“Customize Existing Bug Finder Report Template”
“Fix Insufficient Memory Errors During Polyspace Report Generation”

# polyspace-results-export

(DOS/UNIX) Export Polyspace results to external formats such as CSV or JSON

## Syntax

```
polyspace-results-export -format exportFormat -results-dir resultsFolder [
export options]
```

```
polyspace-results-export -format exportFormat -host hostName -run-id runID [
export options] [polyspace access options]
```

## Description

**Note** This Polyspace command is available in *polyspaceroot*\polyspace\bin. Here, *polyspaceroot* is the Polyspace installation folder, for instance, C:\Program Files\Polyspace \R2023a (see also “Installation Folder” for desktop products or “Installation Folder” for server products). To avoid typing the full path to this command, add this location to the PATH environment variable in your operating system.

polyspace-results-export -format exportFormat -results-dir resultsFolder [ export options] exports Polyspace analysis results stored locally in resultsFolder to an external format exportFormat.

polyspace-results-export -format exportFormat -host hostName -run-id runID [ export options] [polyspace access options] exports Polyspace analysis results stored in Polyspace Access to an external format exportFormat. Specify the Polyspace Access instance by using hostName and the project on Polyspace Access by using runID.

## Examples

### Export Polyspace Results Stored Locally

Export results from a project myProject in the Polyspace user interface to JSON format.

Suppose your project is stored in C:\Polyspace_Workspace. To export results from a specific module in the project, specify the path to the folder that directly contains results from the module.

```
polyspace-results-export -format json-sarif ^
    -results-dir C:\Polyspace_Workspace\myProject\Module_1\BF_Result ^
    -output-name C:\Polyspace_Workspace\reports\myProject\myProject.json
```

(The ^ is used to continue the command to the next line in Windows. If using a different operating system, use the appropriate character for continuing on to the next line.)

### Export Results Stored in Polyspace Access

**Note** To generate reports of results on Polyspace Access at the command line, you must have a Polyspace Bug Finder Server or Polyspace Code Prover Server installation.

Suppose that you want to export the results of a project on Polyspace Access to JSON format.

To connect to Polyspace Access, provide a host name and your login credentials including your encrypted password. To encrypt your password, use the `polyspace-access` command and enter your user name and password at the prompt.

```
polyspace-access -encrypt-password
login: jsmith
password:
CRYPTED_PASSWORD LAMMMEACDMKEFELKMNDCONEAPECEEKPL
Command Completed
```

Store the login and encrypted password in a credentials file and restrict read and write permission on this file. Open a text editor, copy these two lines in the editor, then save the file as `myCredentials.txt` for example.

```
 -login jsmith
 -encrypted-password LAMMMEACDMKEFELKMNDCONEAPECEEKPL
```

To restrict the file permissions, right-click the file and select the **Permissions** tab on Linux or the **Security** tab on Windows.

To specify project results on Polyspace Access, specify the run ID of the project. To obtain a list of projects with their latest run IDs, use the `polyspace-access` with option `-list-project`.

```
polyspace-access -host myAccessServer -credentials-file myCredentials.txt -list-project
Connecting to https://myAccessServer:9443
Connecting as jsmith
Get project list with the last Run Id
Restricted/Code_Prover_Example (Code Prover) RUN_ID 14
public/Bug_Finder_Example (Bug Finder) RUN_ID 24
public/CP/Code_Prover_Example (Polyspace Code Prover) RUN_ID 16
public/Polyspace (Code Prover) RUN_ID 28
Command Completed
```

For more information on this command, see `polyspace-access`.

If Polyspace Access uses the HTTPS protocol, use the `polyspace-report-generator` binary to configure `polyspace-results-export` to enable communications with Polyspace Access over HTTPS. See “Configure Report Generator with Client Keystore” on page 4-53.

Export results from the project with run ID 16 to JSON format.

```
polyspace-results-export -credentials-file myCredentials.txt ^
 -format json-sarif ^
 -host myAccessServer ^
 -run-id 16
```

(The ^ is used to continue the command to the next line in Windows. If using a different operating system, use the appropriate character for continuing on to the next line.)

## Input Arguments

### `exportFormat` — Output format for results

`csv` | `json-sarif` | `console`

Format in which the Polyspace results are exported: `csv` for tab separated values (TSV) output or `json-sarif` for JSON output. If you use the `polyspace-bug-finder-access` command for single-file analysis (Polyspace as You Code), you can also export the results to the console using the format `console`.

Each result consists of information such as result name, family, and so on. Both TSV and JSON formats result in almost the same content being exported but the exported content might refer to the same type of information by different names.

- In the TSV format, each result consists of tab-separated information in columns such as `ID`, `Family`, `Group`, `Color`, `Check`, and so on.

  To package and potentially filter your result data, use the `csv` format. For instance, you can import the TSV file to Microsoft Excel® and use Excel filters on the results.
- In the JSON format, each result consists of almost the same information as JSON object properties. The properties shown for a result sometimes use a name that is different from the name used in the CSV format. For instance, to get the full rule checker name for a result, use the `ruleId` property of a result in combination with the `id` and `name` property of a rule. The reason for the difference is that the JSON format follows the standard notation provided by the OASIS Static Analysis Results Interchange Format (SARIF).

  The JSON format contains some additional information such as the checker short name and the full message that accompanies a result. Use the JSON format if you want to use this short name or message. You can also use this format for a more standardized reporting of results. For instance, if you use several static analysis tools and want to report their results in one interface by using a single parsing algorithm, you can export all the results to the standard SARIF JSON format.

The console output is preformatted in a form similar to compiler errors and warnings, and contains less information than the other formats. In particular, if you baseline Polyspace as You Code results using integration results in the Polyspace Access web interface, use the JSON or CSV format for maximum benefits from the baselining. See “Baseline Polyspace as You Code Results on Command Line”.

### `resultsFolder` — Result folder path

string

Path to a folder containing Polyspace analysis results (`.psbf` or `.pscp` file). If you do not specify a folder path, the command generates a report for analysis results in the current folder.

Example: `C:\Polyspace_Workspace\My_project\Module_1\results`

### `hostName` — Polyspace Access machine host name

string

`hostName` corresponds to the host name that you specify in the URL of the Polyspace Access interface, for example `https://`*`hostName`*`:`*`port`*`/metrics/index.html`. If you are unsure about which host name to use, contact your Polyspace Access administrator. The default host name is `localhost`. You must specify a host name to generate a report for results on the Polyspace Access database.

Example: `my-company-server`

### `runID` — Polyspace Access run ID

integer

Run ID of the project findings that you export. A unique run ID is assigned to each analysis run that you upload to Polyspace Access.

You can see the run ID of a project in the Polyspace Access web interface. To get the run ID of a project at the command line, use the command `polyspace-access` with option `-list-project`.

### `export options` — Additional options for exporting results

string

| Option | Description |
|---|---|
| `-output-name` *`outputName`* | Name of the exported file. The default name is `results_list.txt` (TSV) or `results_list.json` (JSON).<br>The file is saved on the path from which you call the command. To save the file in a different folder, specify the full path to the folder, for instance `-output-name C:\PathTo\OtherFolder`.<br>This option is not compatible with the console output format (`-format console`). |
| `-set-language-english` | Use this option if your display language is set to a language other than English but you want the exported results in English. |
| `-key-mode file-scope \| function-scope` | Specify how the entries in the **Key** column of the exported results are calculated:<br>• `file-scope` (default) — The key is calculated from the result name, result type, and file location.<br>• `function-scope` — The key is calculated from the result name, result type, and function location if the results is located inside a function, or file location otherwise. Keys generated in this mode are prefixed with `FN` if the result is inside a function.<br>See also “Enable Function Scope for Exported Keys”. |

To see options available with this command, enter `polyspace-results-export -h`.

### `polyspace access options` — Additional options for exporting results from Polyspace Access

string

| Option | Description |
|---|---|
| `-port` *`portNumber`* | `portNumber` corresponds to the port number that you specify in the URL of the Polyspace Access interface, for example `https://`*`hostname`*`:`*`portNumber`*`/metrics/index.html`. If you are unsure about which port number to use, contact your Polyspace Access administrator. The default port number is `9443`. |
| `-protocol http \| https` | HTTP protocol to connect to Polyspace Access. Default value is `https`. |
| `-credentials-file` *`file_path`* | Full path to the text file where you store your login credentials. Use this option if, for instance, you use a command that requires your Polyspace Access credentials in a script but you do not want to store your credentials in that script. While the script runs, someone inspecting currently running processes cannot see your credentials.<br>You can store only one set of credentials in the file, either as `-login` and `-encrypted-password` entries on separate lines, for instance:<br>`-login jsmith`<br>`-encrypted-password`<br>`  LAMMMEACDMKEFELKMNDCONEAPECEEKPL`<br>or as an `-api-key` entry:<br>`-api-key keyValue123`<br>Make sure that you restrict the read and write permissions on the file where you store your credentials. For example, to restrict read and write permissions on file `login.txt` in Linux, use this command:<br>`chmod go-rwx login.txt` |

| Option | Description |
|---|---|
| `-api-key` *`keyValue`* | API key you use as a login credential instead of providing your login and encrypted password. To assign an API key to a user, see “Configure User Manager” or contact your Polyspace Access administrator.<br>Use the API key if, for instance, you use a command that requires your Polyspace Access login credentials as part of an automation script with a CI tool like Jenkins. If a user updates his or her password, you do not need to update the API key associated with that user in your scripts.<br>It is recommended that you store the API key in a text file and pass that file to the command by using `-credentials-file`. See the description for option `-credentials-file`. |
| `-login` *`username`*<br>`-encryted-password` *`ENCRYPTED_PASSWD`* | Credentials that you use to log into Polyspace Access. The argument of `-encrypted-password` is the output of the `polyspace-access -encrypt-password` command.<br>For more information on the command, see `polyspace-access`. |

## Version History

**Introduced in R2020b**

### R2023a: CWE ID column is removed

The column that lists the CWE rules that are mapped to Bug Finder defects is removed. Polyspace supports the CWE standard natively, and when you export analysis results, violations of CWE rules are reported in the same way as violations of other coding standards, one results per line. See also “Common Weakness Enumeration (CWE)”.

## See Also

`polyspace-report-generator`

# polyspace-comments-import

(DOS/UNIX) Import review information from previous Polyspace analysis

## Syntax

```
polyspace-comments-import -diff-rte prevResultsFolder currentResultsFolder [-print-new-results] [-overwrite-destination-comments]
```

## Description

**Note** This Polyspace command is available in *`polyspaceroot`*`\polyspace\bin`. Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a` (see also “Installation Folder” for desktop products or “Installation Folder” for server products). To avoid typing the full path to this command, add this location to the `PATH` environment variable in your operating system.

`polyspace-comments-import -diff-rte prevResultsFolder currentResultsFolder [-print-new-results] [-overwrite-destination-comments]` imports review information from a results file in `prevResultsFolder` to `currentResultsFolder`. The review information includes the severity, status and additional notes that you assign to a result.

Besides importing the review information, the command also shows the number of results where review information could not be imported either because the result changed or the result already had new review information. If you use the option `-print-new-results`, you see this information:

- Number of new results in current results folder, that is, results not present in previous results folder
- Number of results removed from previous results folder, that is, results no longer present in the current results folder
- Number of results in current results folder that do not have associated review information

You can also use this command to create a baseline for the analysis results. In the Polyspace user interface, if you click the **New** button, only the analysis results that are new compared to the baseline remain in the results list.

If you use the option `-overwrite-destination-comments`, newer review information on previously existing results are overwritten with previous review information. For instance, if the same result has a different status in the current and previous results folder, after using the `polyspace-comments-import` command:

- Without the option, the result in the current results folder retains its status.
- With the option, the status of the result in the current results folder is overwritten with the status from the previous results folder.

## Examples

### Import Review Information from Previous Polyspace Results

Run Bug Finder on a sample file and add some review information. Then, run Bug Finder a second time and import the information from the previous run.

Copy the file `numerical.c` from *`polyspaceroot`*`\polyspace\examples\cxx\Bug_Finder_Example\sources` to a writable folder. Open a command window and navigate to the folder (using `cd`). Run Bug Finder on the file and save results in the subfolder `Run_1`:

```
polyspace-bug-finder -sources numerical.c -results-dir Run_1/
```

Depending on the product installed, you can also run `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server`.

Open the results file in the `Run_1` subfolder:

```
polyspace Run_1/ps_results.psbf
```

Select a result. On the **Result Details** window, select a **Severity** and **Status** and add some notes. You will import this review information to results from a later analysis.

Run Bug Finder again, but save the results in a different subfolder `Run_2`:

```
polyspace-bug-finder -sources numerical.c -results-dir Run_2/
```

You can open the results file in `Run_2` and see that there is no review information.

Import the review information from the results file in the `Run_1` subfolder to the `Run_2` subfolder. Add the option `-print-new-results` to see the number of new, removed and unreviewed results.

```
polyspace-comments-import -diff-rte Run_1/ Run_2/ -print-new-results
```

Open the results file in the `Run_2` subfolder:

```
polyspace Run_2/ps_results.psbf
```

You see the review information imported from the results file in the `Run_1` subfolder.

## Input Arguments

### `prevResultsFolder` — Folder containing previous Polyspace results with review information

string

Path to a folder containing a Polyspace results file (`.psbf` file for Bug Finder results and `.pscp` file for Code Prover results). The results are presumably from an earlier Polyspace analysis and contain review information that will be imported to a later results file.

Example: `"C:\Polyspace\Project_1_Run_25"`

### `currentResultsFolder` — Folder containing later Polyspace results

string

Path to a folder containing Polyspace results (`.psbf` file for Bug Finder results and `.pscp` file for Code Prover results). The results are presumably from a later Polyspace analysis and have no review information or review information for new results only. You want to import review information from an earlier Polyspace analysis to these results.

Example: `"C:\Polyspace\Project_1_Run_26"`

## Version History

**Introduced in R2013b**

## See Also

`-import-comments`

### Topics

“Import Review Information from Previous Polyspace Analysis”
“Import Review Information from Existing Polyspace Access Projects”

# MATLAB and Simulink Functions, Classes, and Methods

5

# Functions, Properties, Classes, and Apps

# pslinkfun

Manage model analysis at the command line

## Syntax

```
pslinkfun('annotations','type',typeValue,'kind',kindValue,Name,Value)
```

```
pslinkfun('openresults',systemName)
```

```
pslinkfun('settemplate',psprjFile)
prjTemplate = pslinkfun('gettemplate')
```

```
pslinkfun('advancedoptions')
pslinkfun('enablebacktomodel')
pslinkfun('help')
pslinkfun('metrics')
pslinkfun('jobmonitor')
pslinkfun('stop')
```

## Description

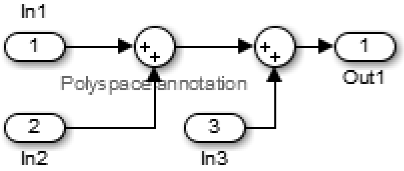
`pslinkfun('annotations','type',typeValue,'kind',kindValue,Name,Value)` adds an annotation of type `typeValue` and kind `kindValue` to the selected block in the model. You can specify a different block using a `Name,Value` pair argument. You can also add notes about a severity classification, an action status, or other comments using `Name,Value` pairs.

In the generated code associated with the annotated block, Polyspace adds code comments before and after the lines of code. Polyspace reads these comments and marks Polyspace results of the specified `kind` with the annotated information.

Syntax limitations:

- You can have only one annotation per block. If a block produces both a rule violation and an error, you can annotate only one type.
- Even though you apply annotations to individual blocks, the scope of the annotation can be larger. The generated code from one block can overlap with another, causing the annotation to also overlap.

  For example, consider this model. The first summation block has a Polyspace annotation, but the second does not.

  However, the associated generated code adds all three inputs in one line of code.

```
/* polyspace:begin<RTE:OVFL:Medium:To Fix>*/
annotate_y.Out1=(annotate_u.In1+annotate_U.In2)+annotate_U.In3;
/* polyspace:end<RTE:OVFL:Medium:To Fix> */
```

Therefore, the annotation justifies both summations.

`pslinkfun('openresults',systemName)` opens the Polyspace results associated with the model or subsystem `systemName` in the Polyspace environment.

`pslinkfun('settemplate',psprjFile)` sets the configuration file for new verifications.

`prjTemplate = pslinkfun('gettemplate')` returns the template configuration file used for new analyses.

`pslinkfun('advancedoptions')` opens the advanced verification options window to configure additional options for the current model.

`pslinkfun('enablebacktomodel')` enables the back-to-model feature of the Simulink plug-in. If your Polyspace results do not properly link to back to the model blocks, run this command.

`pslinkfun('help')` opens the Polyspace documentation in a separate window. Use this option for only pre-R2013b versions of MATLAB.

`pslinkfun('metrics')` opens the Polyspace Metrics interface.

`pslinkfun('jobmonitor')` opens the Polyspace Job Monitor to display remote verifications in the queue.

`pslinkfun('stop')` kills the code analysis that is currently running. Use this option for local analyses only.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

## Examples

### Annotate a Block and Run a Polyspace Bug Finder Analysis

Use the Polyspace annotation function to annotate a block and see the annotation in the analysis results.

In the example model WhereAreTheErrors, add an annotation to the switch block for MISRA C rule 13.7 violations with a comment, a severity, and a status.

```
model = 'WhereAreTheErrors';
open(model)
pslinkfun('annotations','type','Misra-C', 'kind', '13.7','block',...
    'WhereAreTheErrors/Switch1','status','to fix','comment','must fix')
```

In the open model, you can see a Polyspace annotation added to the Switch block.

Generate code for the model and run an analysis. After the analysis is finished, open the results in the Polyspace environment:

```
slbuild(model)
opts = pslinkoptions(model);
opts.VerificationMode = 'BugFinder';
opts.VerificationSettings = 'PrjConfigAndMisra';
pslinkrun(model,opts)
pslinkfun('openresults',model)
```

The five MISRA C 13.7 rule violations are annotated with the information you added to the switch block. The annotations appear in the **Status** and **Comment** columns.

### Add Batch Options to Default Configuration Template

Change advanced Polyspace options and set the new configuration as a template.

Load the model `WhereAreTheErrors` and open the advanced options window.

```
model = 'WhereAreTheErrors';
load_system(model)
pslinkfun('advancedoptions')
```

In the **Run Settings** pane, select the options **Run Bug Finder analysis on a remote cluster** and **Upload results to Polyspace Metrics**.

Set the configuration template for new Polyspace analyses to have these options.

```
pslinkfun('settemplate',fullfile(cd,'pslink_config',...
    'WhereAreTheErrors_config.psprj'))
```

View the current Polyspace template.

```
template = pslinkfun('gettemplate')
```

```
template =
C:\ModelLinkDemo\pslink_config\WhereAreTheErrors_config.psprj
```

### View Polyspace Queue and Metrics

Run a remote analysis, view the analysis in the queue, and review the metrics.

Before performing this example, check that your Polyspace configuration is set up for remote analysis and Polyspace Metrics.

Build the model `WhereAreTheErrors`, create a Polyspace options object, set the verification mode, and open the advanced options window.

```
model = 'WhereAreTheErrors';
load_system(model)
slbuild(model)
opts = pslinkoptions(model);
opts.VerificationMode = 'BugFinder';
pslinkfun('advancedoptions')
```

In the **Run Settings** pane, select the options **Run Bug Finder analysis on a remote cluster** and **Upload results to Polyspace Metrics**.

Run Polyspace, then open the Job Monitor to monitor your remote job.

```
pslinkrun(model,opts)
pslinkfun('jobmonitor')
```

After your job is finished, open the metrics server to see your job in the repository.

```
pslinkfun('metrics')
```

## Input Arguments

### typeValue — type of result

'DEFECT' | 'MISRA-C' | 'MISRA-AC-AGC' | 'MISRA-CPP' | 'JSF'

The type of result with which to annotate the block, specified as:

- 'DEFECT' for defects.
- 'MISRA-C' for MISRA C coding rule violations (C code only).
- 'MISRA-AC-AGC' for MISRA C coding rule violations (C code only).
- 'MISRA-CPP' for MISRA C++ coding rule violations (C++ code only).
- 'JSF' for JSF C++ coding rule violations (C++ code only).

Example: 'type','MISRA-C'

### kindValue — specific check or coding rule

check acronym | rule number

The specific check or coding rule specified by the acronym of the check or the coding rule number. For the specific input for each type of annotation, see the following table.

| type Value | kind Values |
|---|---|
| 'DEFECT' | Use the abbreviation associated with the type of defect that you want to annotate. For example, 'int_ovfl' – Integer overflow.<br>For the list of possible checks, see: “Complete List of Polyspace Bug Finder Results”. |
| 'MISRA-C' | Use the rule number that you want to annotate. For example, '2.2'.<br>For the list of supported MISRA C rules and their numbers, see “MISRA C:2004 and MISRA AC AGC Coding Rules”. |
| 'MISRA-AC-AGC' | Use the rule number that you want to annotate. For example, '2.2'.<br>For the list of supported MISRA C rules and their numbers, see “MISRA C:2004 and MISRA AC AGC Coding Rules”. |
| 'MISRA-CPP' | Use the rule number that you want to annotate. For example, '0-1-1'.<br>For the list of supported MISRA C++ rules and their numbers, see “MISRA C++:2008 Rules”. |

| type Value | kind Values |
|---|---|
| 'JSF' | Use the rule number that you want to annotate. For example, '3'.<br>For the list of supported JSF C++ rules and their numbers, see "JSF AV C ++ Coding Rules". |

Example: pslinkfun('annotations','type','MISRA-CPP','kind','1-2-3')

Data Types: char

**systemName — Simulink model**
system | subsystem

Simulink model specified by the system or subsystem name.

Example: pslinkfun('openresults','WhereAreTheErrors')

**psprjFile — Polyspace project file**
standard Polyspace template (default) | absolute path to .psprj file

Polyspace project file specified as the absolute path to the .psprj project file. If psprjFile is empty, Polyspace uses the standard Polyspace template file. New Polyspace projects start with this project configuration.

Example: pslinkfun('settemplate', fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', 'Bug_Finder_Example', 'Bug_Finder_Example.bf.psprj'));

### Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose* Name *in quotes.*

Example: 'block','MyModel\Sum', 'status','to fix'

**block — block to be annotated**
gcb (default) | block name

The block you want to annotate specified by the block name. If you do not use this option, the block returned by the function gcb is annotated.

Example: 'block','MyModel\Sum'

**class — severity of the check**
'high' | 'medium' | 'low' | 'unset'

Severity of the check specified as high, medium, low, or unset.

Example: 'class','high'

**status — action status**
'unreviewed' | 'to investigate' | 'to fix' | 'justified' | 'no action planned' | 'not a defect' | 'other'

Action status of the check specified as unreviewed, to investigate, to fix, justified, no action planned, not a defect, or other.

Example: `'status','no action planned'`

**`comment` — additional comments**
character vector

Additional comments specified as a character vector. The comments provide more information about why the results are justified.

Example: `'comment','defensive code'`

## Version History

**Introduced in R2014a**

### See Also

`pslinkrun` | `pslinkoptions` | `gcb`

# pslinkoptions

Create an options object to customize configuration of a Simulink model, generated code or a S-Function block. Use the object to specify configuration options for these Simulink objects in a Polyspace run from the MATLAB command line

## Syntax

```
opts = pslinkoptions(codegen)
opts = pslinkoptions(model)
opts = pslinkoptions(sfunc)
```

## Description

`opts = pslinkoptions(codegen)` returns an options object with the configuration options for code generated by `codegen`.

`opts = pslinkoptions(model)` returns an options object with the configuration options for the Simulink model.

`opts = pslinkoptions(sfunc)` returns an options object with the configuration options for the S-Function.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

## Examples

### Create generic options object for code generated by Embedded Coder

This example shows how you can create a generic Polyspace options object that is suitable for analyzing code generated by using Embedded Coder. This options object is prepopulated with appropriate Embedded Coder parameters. Edit the options object to modify the generic analysis.

Create a new Polyspace configuration object `new_opt` by calling the function `pslinkoptions` and specify `'ec'` as the input argument.

```
new_opt = pslinkoptions('ec')
```

```
new_opt = 

                    ResultDir: 'results_$ModelName$'
         VerificationSettings: 'PrjConfig'
           OpenProjectManager: 0
         AddSuffixToResultDir: 0
     EnableAdditionalFileList: 0
           AdditionalFileList: {}
             VerificationMode: 'CodeProver'
          EnablePrjConfigFile: 0
```

```
                PrjConfigFile: ''
         AddToSimulinkProject: 0
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
              OutputRangeMode: 'None'
           ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
                  AutoStubLUT: 1
      CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
        VerifAllSFcnInstances: 0
```

By default, this options object uses the same verification settings that you specify in the Polyspace project. To check MISRA C® 2012 coding rule violations in addition to the existing verifications specified in the project, run this code at the MATLAB command line:

```
new_opt.VerificationSettings = 'PrjConfigAndMisraC2012'
```

```
new_opt = 

                    ResultDir: 'results_$ModelName$'
         VerificationSettings: 'PrjConfigAndMisraC2012'
           OpenProjectManager: 0
         AddSuffixToResultDir: 0
     EnableAdditionalFileList: 0
           AdditionalFileList: {}
             VerificationMode: 'CodeProver'
          EnablePrjConfigFile: 0
                PrjConfigFile: ''
         AddToSimulinkProject: 0
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
              OutputRangeMode: 'None'
           ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
                  AutoStubLUT: 1
      CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
        VerifAllSFcnInstances: 0
```

When you start the Polyspace analysis of the generated code, you might want to open the Polyspace User Interface to follow the progress of the and to review the results afterwards. To open the Polyspace interface when you start the analysis, run this code:

```
new_opt.OpenProjectManager = true
```

```
new_opt = 

                    ResultDir: 'results_$ModelName$'
         VerificationSettings: 'PrjConfigAndMisraC2012'
           OpenProjectManager: 1
         AddSuffixToResultDir: 0
     EnableAdditionalFileList: 0
           AdditionalFileList: {}
             VerificationMode: 'CodeProver'
          EnablePrjConfigFile: 0
                PrjConfigFile: ''
         AddToSimulinkProject: 0
```

```
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
              OutputRangeMode: 'None'
           ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
                  AutoStubLUT: 1
      CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
        VerifAllSFcnInstances: 0
```

### Create and edit options object to modify Polyspace configuration

This example shows how you can store the Polyspace configurations of a Simulink model in to an object, and use the object to edit the configuration options.

Load the model `closed_loop_control`.

```
load_system('closed_loop_control');
```

To create an object containing the Polyspace configurations of the model, call `pslinkoptions`.

```
model_opt = pslinkoptions('closed_loop_control')
```

```
model_opt = 

                    ResultDir: 'results_$ModelName$'
         VerificationSettings: 'PrjConfig'
           OpenProjectManager: 0
         AddSuffixToResultDir: 0
     EnableAdditionalFileList: 0
           AdditionalFileList: {}
             VerificationMode: 'CodeProver'
          EnablePrjConfigFile: 0
                PrjConfigFile: ''
         AddToSimulinkProject: 0
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
              OutputRangeMode: 'None'
           ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
                  AutoStubLUT: 1
      CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
        VerifAllSFcnInstances: 0
```

The model is already configured for Embedded Coder®, so only the Embedded Coder configuration options appear as the fields of the object `model_opt`.

To modify a Polyspace configuration option, set the corresponding field of `model_opt`. For instance, change the results directory and set the verification mode to `CodeProver` by modifying fields : `model_opt.ResultDir` and `model_opt.VerificationMode`, respectively.

```
model_opt.ResultDir = 'results_v1_$ModelName$';
model_opt.VerificationMode = 'CodeProver'
```

```
model_opt = 

                    ResultDir: 'results_v1_$ModelName$'
         VerificationSettings: 'PrjConfig'
           OpenProjectManager: 0
         AddSuffixToResultDir: 0
     EnableAdditionalFileList: 0
           AdditionalFileList: {}
             VerificationMode: 'CodeProver'
          EnablePrjConfigFile: 0
                PrjConfigFile: ''
         AddToSimulinkProject: 0
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
              OutputRangeMode: 'None'
           ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
                  AutoStubLUT: 1
      CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
        VerifAllSFcnInstances: 0
```

### Create and edit an options object for TargetLink at the command line

Create a Polyspace® options object called `new_opt` with TargetLink® parameters:

```
new_opt = pslinkoptions('tl')
```

```
new_opt = 

                   ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 0
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {}
            VerificationMode: 'CodeProver'
         EnablePrjConfigFile: 0
               PrjConfigFile: ''
        AddToSimulinkProject: 0
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
                 AutoStubLUT: 1
```

Set the `OpenProjectManager` option to true to follow the progress in the Polyspace interface. Also change the configuration to check for both run-time errors and MISRA C® coding rule violations:

```
new_opt.OpenProjectManager = true;
new_opt.VerificationSettings = 'PrjConfigAndMisra'
```

```
new_opt = 

                   ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfigAndMisra'
          OpenProjectManager: 1
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {}
            VerificationMode: 'CodeProver'
         EnablePrjConfigFile: 0
               PrjConfigFile: ''
        AddToSimulinkProject: 0
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
                 AutoStubLUT: 1
```

## Input Arguments

### codegen — Code generator
'ec' | 'tl'

Code generator, specified as either `'ec'` for Embedded Coder or `'tl'` for TargetLink®. Each argument creates a Polyspace options object with properties specific to that code generator.

For a description of all configuration options and their values, see pslinkoptions.

Example: `ec_opt = pslinkoptions('ec')`

Example: `tl_opt = pslinkoptions('tl')`

Data Types: `char`

### model — Simulink model name
model name

Simulink model, specified by the model name. Creates a Polyspace options object with the configuration options of that model. If you have not set any options, the object has the default configuration options. If you have set a code generator, the object has the default options for that code generator.

For a description of all configuration options and their values, see pslinkoptions.

Example: `model_opt = pslinkoptions('my_model')`

Data Types: `char`

### sfunc — path to S-Function
character vector

Path to S-Function, specified as a character vector. Creates a Polyspace options object with the configuration options for the S-function. If you have not set any options, the object has the default configuration options.

For a description of all configuration options and their values, see pslinkoptions.

Example: `sfunc_opt = pslinkoptions('path/to/sfunction')`

Data Types: `char`

## Output Arguments

**`opts` — Polyspace configuration options**
options object

Polyspace configuration options, returned as an options object. The object is used with `pslinkrun` to run Polyspace from the MATLAB command line.

For the list of object properties, see pslinkoptions.

Example: `opts= pslinkoptions('ec')`
`opts.VerificationSettings = 'Misra'`

# Version History

**Introduced in R2012a**

## See Also

`pslinkrun` | `pslinkfun` | pslinkoptions

# polyspacesetup

Integrate Polyspace installation with Simulink

## Syntax

```
polyspacesetup('install')
polyspacesetup('install', 'polyspacefolder', folder)
polyspacesetup('install', 'polyspacefolder', folder, 'silent', isSilent)
polyspacesetup('uninstall')
polyspacesetup('showpolyspacefolders')
```

## Description

`polyspacesetup('install')` integrates Polyspace from the default installation folder with MATLAB or Simulink. If you installed Polyspace in a nondefault folder, the function prompts you for the installation folder. See “Installation Folder”.

To run MATLAB scripts for Polyspace analysis, install MATLAB and Polyspace in separate folders, and then integrate them by using this function. See “Integrate Polyspace with MATLAB and Simulink” or “Integrate Polyspace Server Products with MATLAB”.

`polyspacesetup('install', 'polyspacefolder', folder)` integrates Polyspace installed in the folder `folder` with MATLAB or Simulink.

`polyspacesetup('install', 'polyspacefolder', folder, 'silent', isSilent)` integrates Polyspace installed in the folder `folder` with MATLAB or Simulink silently when `isSilent` is `true` or with a prompt if `isSilent` is `false`. When you start MATLAB with the option `-batch`, `isSilent` is set to `true` by default. If you use a nondefault folder to install Polyspace and then do not specify the folder in `folder`, you are prompted to specify the install location even if you use `-batch` to start MATLAB.

`polyspacesetup('uninstall')` unlinks the currently linked installation of Polyspace from MATLAB.

`polyspacesetup('showpolyspacefolders')` lists all Polyspace folders that are linked to your current installation of MATLAB.

## Examples

### Integrate Polyspace with MATLAB or Simulink

To integrate Polyspace with MATLAB or Simulink, use the function `polyspacesetup`.

Open MATLAB with administrator or root privilege.

At the MATLAB command prompt, enter:

```
polyspacesetup('install');
```

If you install Polyspace in the default folder `C:\Program Files\Polyspace\R2023a`, the command integrates Polyspace with MATLAB. You might be prompted that the workspace will be cleared and all open models closed. Click **Yes**. The process might take a few minutes to complete. When you start MATLAB with the `-batch` option, the installation completes without any prompts .

If a Polyspace installation is not detected at the default location, you are prompted for the installation location. Use this command:

```
polyspacesetup('install', 'polyspaceFolder', Folder)
```

where *`Folder`* is the Polyspace installation folder.

Restart MATLAB.

### Integrate Polyspace Noninteractively with MATLAB at Command Line by Using `-batch`

To integrate Polyspace with MATLAB in the command line noninteractively, start MATLAB with the startup option `-batch`. See “Commonly Used Startup Options”.

When you start MATLAB with the startup option `-batch`, `polyspacesetup` is silent by default. That is, the function does not emit any messages unless there is any input error.

1. Open a Windows command-line prompt with administrator or root privilege.
2. To ensure that the integration takes place noninteractively, specify the install folder for Polyspace. At the command-line prompt, enter:

   ```
   matlab -batch "polyspacesetup('install','polyspaceFolder',folder)"
   ```

   where *`folder`* is the installation location of Polyspace.
3. If the integration is successful, this message is displayed:

   ```
   Polyspace plug-in: installation complete.
   Restart MATLAB before using Polyspace plug-in.
   ```

   You can also enter the command in a script. For instance, you might have a script that performs the installations of MATLAB and Polyspace. Append the preceding command to your script to integrate MATLAB and Polyspace noninteractively.

### Silently Integrate Polyspace with MATLAB or Simulink

To integrate Polyspace with MATLAB or Simulink silently, use the function `polyspacesetup`. By default, Polyspace is installed in the folder `C:\Program Files\Polyspace\R2023a`.

Open MATLAB with administrator or root privilege.

At the MATLAB command prompt, enter:

```
polyspacesetup('install', 'polyspaceFolder', Folder, 'silent', true);
```

where *`Folder`* is the Polyspace installation folder. The process might take a few minutes to complete.

Restart MATLAB.

## Input Arguments

### `folder` — Polyspace installation folder

Path to Polyspace installation

Path to the Polyspace installation folder specified as a character vector.

Example: `'C:\Program Files\Polyspace\R2023a'`

Data Types: `char`

### `isSilent` — Indicator for silent integration

`false` (default) | `true`

Whether to perform the integration silently (without prompting for user input), specified as `true` or `false`. When you start MATLAB with the `-batch` option, this argument is set to `true` by default .

Data Types: `logical`

# Version History

**Introduced in R2019a**

## See Also

`polyspace.Project`

### Topics

"Run Polyspace Analysis by Using MATLAB Scripts"
"Integrate Polyspace Server Products with MATLAB"

# polyspacePackNGo

Generate and package options files to run Polyspace analysis on code generated from Simulink model

## Syntax

```
archivePath = polyspacePackNGo(mdlName)
archivePath = polyspacePackNGo(mdlName,psOpt)
archivePath = polyspacePackNGo(mdlName,psOpt,asModelRef)
```

## Description

`archivePath = polyspacePackNGo(mdlName)` examines the Simulink model `mdlName`, extracts Polyspace options files from it, and packages the options files in the zip file located at `archivePath`. Before using `polyspacePackNGo`, generate code from your Simulink model. Then archive the generated code, for instance, by using `packNGo`. Generate the Polyspace options files from the Simulink model and include them in the code archive by using `polyspacePackNGo`. In a different development environment, when running a Polyspace analysis of the generated code, use the options files included in the code archive to preserve model-specific information, such as design range specifications. You must have Embedded Coder to use `slbuild`.

`archivePath = polyspacePackNGo(mdlName,psOpt)` generates and packages the Polyspace options files that are generated according to the specification in `psOpt`. The object `psOpt` must be a Polyspace options object that is generated by using `pslinkoptions`. Using `psOpt`, modify the options for the Polyspace analysis.

`archivePath = polyspacePackNGo(mdlName,psOpt,asModelRef)` generates and packages the Polyspace options files by using `asModelRef` to specify whether to generate option files for model reference code or standalone code.

## Examples

### Generate and Package Polyspace Options Files

To generate and package Polyspace options files for a Simulink model, use `polyspacePacknGo`.

Open the Simulink model `rtwdemo_counter` and specify a folder for storing the generated code.

```
% Make temporary folders for code genration
[TEMPDIR, CGDIR] = rtwdemodir();
% Open the model
mdlName = 'rtwdemo_counter';
open_system(mdlName);
% Specify a folder for generated code
codegenFolder = 'rtwdemo_counter_ert_rtw';
```

To enable packing the generated code in an archive, set the option `PackageGeneratedCodeAndArtifacts` to `true`. Specify the name of the generated code archive as `genCodeArchive.zip`.

```
configSet = getActiveConfigSet(mdlName);
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true);
set_param(configSet, 'PackageName', 'genCodeArchive.zip');
```

To make the model compatible with Polyspace, set `SystemTargetFile` to `ert.tlc`.

```
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

After configuring the model, generate code.

```
slbuild(mdlName)
```

Because `PackageGeneratedCodeAndArtifacts` is set to `true`, the generated code is packed into the archive `genCodeArchive.zip`.

Generate and package Polyspace options files.

```
zipFile = polyspacePackNGo(mdlName);
```

In the code archive `genCodeArchive.zip`, the Polyspace options files are packaged in the `polyspace` folder.

### Package Polyspace Options Files That Have Specific Polyspace Analysis Options

To specify the Polyspace analysis options when packaging and generating options files, use `pslinktoptions`.

Open the Simulink model `rtwdemo_counter` and configure the model for generating a code archive that is compatible with Polyspace.

```
% Make temporary folders for code genration
[TEMPDIR, CGDIR] = rtwdemodir();
% Open the model
mdlName = 'rtwdemo_counter';
open_system(mdlName);
% Specify a folder for generated code
codegenFolder = 'rtwdemo_counter_ert_rtw';
configSet = getActiveConfigSet(mdlName);
% Enable packing the generated code into an archive
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true);
% Specify a name for the code archive
set_param(configSet, 'PackageName', 'genCodeArchive.zip');
% Configure the model to be Polyspace Compatible
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

After configuring the model, generate code.

```
slbuild(mdlName)
```

Because `PackageGeneratedCodeAndArtifacts` is set to `true`, the generated code is packed into the archive `genCodeArchive.zip`.

To specify the model configuration for the Polyspace analysis, use a `pslinkoptions` object. Create this object by using the function `pslinkoptions`.

```
psOpt = pslinkoptions(mdlName);
```

The object psopt is a structure where the fields are model configurations that you can specify.

Specify the model configuration by using psOpt object. For instance, set InputRangeMode to full range. For a full options list, see the input argument psOpt.

```
psOpt.InputRangeMode = 'FullRange';
```

Generate and package Polyspace options files. Use the psOpt object as the second argument in polyspacePacknGo.

```
zipFile = polyspacePackNGo(mdlName,psOpt);
```

In the code archive genCodeArchive.zip, the Polyspace options files are packaged in the polyspace folder. The file optionsFile.txt contains the specified Polyspace analysis options.

### Package Polyspace Options Files for Code Generated as a Model Reference

To accelerate model simulations, invoke referenced Simulink models as simulation targets. To generate model reference simulation targets from a Simulink model, generate code from the model by using slbuild with the build process specified as ModelReferenceCoderTarget. Then, package the generated code by using packNGo. To generate and package Polyspace options files for analyzing such code, use the function polyspacePacknGo with the optional argument asModelRef set to true.

Open the Simulink model rtwdemo_counter and configure the model for generating a code archive that is compatible with Polyspace.

```
% Make temporary folders for code genration
[TEMPDIR, CGDIR] = rtwdemodir();
% Load model
mdlName = 'rtwdemo_counter';
load_system(mdlName);
configSet = getActiveConfigSet(mdlName);
% Enable packing the generated code into an archive
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true);
set_param(configSet, 'PackageName', '');
% Configure the model to be Polyspace Compatible
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

After configuring the model, generate a model reference simulation target from it by using the function slbuild. Specify the option ModelReferenceCoderTarget. See slbuild.

```
slbuild(mdlName,'ModelReferenceCoderTarget');
```

The code that is generated is stored in the folder slprj.

To package the code that is generated as a model reference, use the function packNGo. Locate the file buildinfo.mat in <*working folder*>/slprj/ert/rtwdemo_counter and use the full path to it as the input to packNGo. This command generates an archive containing the generated code and the object buildinfo.mat. See packNGo.

```
% Locate buildinfo and generate code archive
buildinfo = fullfile(pwd,'slprj','ert',mdlName,'buildinfo.mat');
packNGo(buildinfo)
```

Generate and package Polyspace options files. Omit the optional second argument. Set the third argument `asModelRef` to `true`.

```
zipFile = polyspacePackNGo(mdlName,[],true);
```

In the code archive `rtwdemo_counter.zip`, the Polyspace options files are packaged in the `polyspace` folder.

## Input Arguments

### `mdlName` — Name of Simulink model for which to generate Polyspace options files

model name

A character array containing the name of the model for which you want to generate and package the Polyspace options files.

Example: `polyspacePackNGo('rtwdemo_roll')`

Data Types: `char`

### `psOpt` — Polyspace options object

options associated with `model` (default) | object created by using `pslinkoptions`

Specifies the model configuration for the Polyspace analysis by using a `pslinkoptions` object. You can modify certain analysis options by modifying `psOpt`, which is a structure where individual fields represent analysis options. For a fill list of options that you can modify, see the table **Polyspace Analysis Options Supported by `polyspacePacknGo`**.

### Polyspace Analysis Options Supported by `polyspacePacknGo`

| Property | Value | Description |
|---|---|---|
| EnableAdditionalFileList: Enable an additional file list to be analyzed, specified as `true` or `false`. Use with the `AdditionalFileList` option. | `true` | Polyspace verifies additional files specified in the `AdditionalFileList` option. |
| | `false` (default) | Polyspace does not verify additional files. |
| AdditionalFileList: List of additional files to be analyzed specified as a cell array of files. To add these files to the analysis, use the `EnableAdditionalFileList` option. | cell array | Polyspace considers the listed files for verification. |
| InputRangeMode: Specifies the range of the input variables. | `'DesignMinMax'` (default) | Polyspace uses the input range defined in the workspace or a block. |
| | `'Fullrange'` | Polyspace uses full range inputs. |
| ParamRangeMode: Specifies the range of the constant parameters. | `'DesignMinmax'` | Polyspace uses the constant parameter range defined in the workspace or in a block. |
| | `'None'` (default) | Polyspace uses the value of parameters specified in the code. |
| OutputRangeMode: Specifies the output assertions. | `'DesignMinMax'` | Polyspace applies assertions to outputs by using a range defined in a block or the workspace. |
| | `'None'` (default) | Polyspace does not apply assertions to the output variables. |
| ModelRefVerifDepth: Specify the depth for analyzing the models that are referenced by the current model. | `'Current model Only'` (default) | Polyspace analyzes only the top model without analyzing the referenced models. Use this option when you refer to models that do not need to be analyzed, such as library models. |

| Property | Value | Description |
|---|---|---|
| | `'1'|'2'|'3'` | Polyspace analyzes referenced models up to the specified depth in the reference hierarchy. To analyze the models that are referenced by the top model, specify the property `ModelRefVerifDepth` as `'1'`. To analyze models that are referenced by the first level of references, specify this property as `'2'`. |
| | `'All'` | Polyspace verifies all referenced models. |
| ModelRefByModelRefVerif: Specify whether you want to analyze all referenced models together or individually. | `true` | Polyspace analyzes the top model and the referenced models together. Use this option to check for integration or scaling issues. |
| | `false` (default) | Polyspace analyzes the top model and the referenced models individually. |
| AutoStubLUT: Specifies how lookup tables are used. | `true` (default) | Polyspace stubs the lookup tables and verifies the model without analyzing the lookup table code. |
| | `false` | Polyspace includes the lookup table code in the analysis. |
| CheckConfigBeforeAnalysis: Specifies the level of configuration checking done before the Polyspace analysis starts. | `'Off'` | Polyspace checks only for errors. The analysis stops if errors are found. |
| | `'OnWarn'` (default) | Polyspace stops the analysis when errors are found and displays a message when warnings are found. |
| | `'OnHalt'` | Polyspace stops the analysis when either errors or warnings are found. |

Example: `polyspacePackNGo('rtwdemo_roll', psOpt)`, where `ps_opt` is an options object created by calling `pslinkoptions`

### `asModelRef` — Indicator for model reference analysis

`false` (default) | `true`

Indicator for model reference analysis, specified as `true` or `false`.

- If `asModelRef` is `false` (default), the function generates options files so that Polyspace analyzes the generated code as standalone code.

- If asModelRef is true, the function generates options files so that Polyspace analyzes the generated code as model reference code.

**Note** If you set asModelRef to true, use slbuild to generate code.

Example: polyspacePackNGo('rtwdemo_roll', psOpt,true)

Data Types: logical

## Output Arguments

**archivePath — The full path to the archive containing the generated options files**
path to archive

A character array containing the path to the generated archive. The options files are located in the polyspace folder in the archive. The polyspace folder contains these options files:

- optionsFile.txt: a text file containing the Polyspace options required to run a Polyspace analysis on the generated code without losing model-specific information, such as design range specification.
- *model*_drs.xml: A file containing the design range specification of the model.
- linksData.xml: A file that links the generated code to the components of the model.

To run a Polyspace analysis on the generated code in an environment that is different than the environment where the code was generated from the Simulink model, use these files.

Data Types: char

# Version History

**Introduced in R2020b**

## See Also

slbuild | pslinkoptions

### Topics

"Run Polyspace Analysis on Generated Code by Using Packaged Options Files"
"Run Polyspace Analysis by Using MATLAB Scripts"
"Integrate Polyspace Server Products with MATLAB"
pslinkoptions Properties

# pslinkrunCrossRelease

Analyze C/C++ code generated by R2020b or newer Embedded Coder versions by using a different version of Polyspace that is more recent than the Simulink version

## Syntax

```
[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem)
[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem,
psOpt)
[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem,
psOpt,asModelRef)
[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem,
psOpt,asModelRef,OptionsFile)
```

## Description

`[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem)` runs a Polyspace analysis of the code generated from `ModelOrSubsystem` by using Embedded Coder from an earlier release of Simulink.

`[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem, psOpt)` runs a Polyspace analysis of the code generated from `ModelOrSubsystem` through an earlier release of Simulink. The analysis uses the model configuration options that are specified in the `pslinkoptions` object `psOpt`.

`[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem, psOpt,asModelRef)` runs a Polyspace analysis of the code generated as a model reference from `ModelOrSubsystem` through an earlier release of Simulink.The analysis uses `asModelRef` to specify which type of generated code to analyze—standalone code or model reference code.

`[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem, psOpt,asModelRef,OptionsFile)` runs a Polyspace analysis of the code generated from `ModelOrSubsystem` through an earlier release of Simulink. The analysis uses the Polyspace analysis options specified in the options file `OptionsFile`.

## Examples

### Analyze Code Generated by Using Earlier Simulink Release

To run a Polyspace analysis on code generated by using an earlier release of Simulink, use the function `pslinkrunCrossRelease`. The analysis uses the configuration options associated with `ModelOrSubsystem`. The Simulink release must be R2020b or later. Before you run an analysis, you must integrate Polyspace with Simulink. See “MATLAB Release Earlier Than Polyspace”.

1 Open the Simulink model `rtwdemo_roll` and configure the model for code generation.

```
% Load the model
model = 'rtwdemo_roll';
load_system(model);
```

```
% Configure the Solver
configSet = getActiveConfigSet(model);
set_param(configSet,'Solver','FixedStepDiscrete');
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

2 To enable packing the generated code in an archive, set the option `PackageGeneratedCodeAndArtifacts` to `true`.

```
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true)
```

3 Create temporary folders for code generation and generate code.

```
[TEMPDIR, CGDIR] = rtwdemodir();
slbuild(model);
```

4 Start a Polyspace analysis.

```
% Run Polyspace analysis
[~,resultsFolder] = pslinkrunCrossRelease(model);
bdclose(model);
```

The character vector `resultsFolder` contains the full path to the results folder.

### Run Polyspace Analysis with Modified Configuration While Analyzing Code Generated by Using Earlier Simulink Release

To run a Polyspace analysis with modified model configurations, use a `pslinkoptions` object. For a list of model configurations related to Polyspace analysis that you can modify, see the table **Polyspace Configuration Parameters Supported by `pslinkrunCrossRelease`** on this page. Before you run an analysis, you must integrate Polyspace with Simulink. See “MATLAB Release Earlier Than Polyspace”.

1 Open the Simulink model `rtwdemo_roll` and configure the model for code generation.

```
% Load the model
model = 'rtwdemo_roll';
load_system(model);
% Configure the Solver
configSet = getActiveConfigSet(model);
set_param(configSet,'Solver','FixedStepDiscrete');
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

2 To enable packing the generated code in an archive, set the option `PackageGeneratedCodeAndArtifacts` to `true`.

```
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true)
```

3 Create temporary folders for code generation and generate code.

```
[TEMPDIR, CGDIR] = rtwdemodir();
slbuild(model);
```

4 To specify the model configurations for the Polyspace analysis, use a `pslinkoptions` object. Create this object by using the function `pslinkoptions`. To run a Bug Finder analysis, set `psOpt.VerificationMode` to `'BugFinder'`. To assert the range defined in a block on its input variables, specify `psOpts.InputRangeMode` as `'DesignMinMax'`.

```
% Create a Polyspace options object from the model.
psOpts = pslinkoptions(model);

% Set model configurtion for the Polyspace analysis.
```

```
psOpts.VerificationMode = 'BugFinder';
psOpts.InputRangeMode = 'DesignMinMax';
```

5. Start a Polyspace analysis. To specify model configuration for the Polyspace analysis, set the object psOpt as the optional second argument in pslinkrunCrossRelease.

   ```
   % Run Polyspace analysis
   [~,resultsFolder] = pslinkrunCrossRelease(model,psOpt);
   bdclose(model);
   ```

   The character vector resultsFolder contains the full path to the results folder.

#### Analyze Code Generated as Model Reference by Using an Earlier Simulink Release

To accelerate model simulations, invoke referenced Simulink models as simulation targets. To generate model reference simulation targets from a Simulink model, generate code from the ModelOrSubsystem by using slbuild with the build process specified as ModelReferenceCoderTarget. Package the generated code by using packNGo. Then, analyze the generated code by running a cross-release Polyspace analysis. Before you run an analysis, you must integrate Polyspace with Simulink. See “MATLAB Release Earlier Than Polyspace”.

1. Open the Simulink model rtwdemo_roll and configure the model for code generation.

   ```
   % Load the model
   model = 'rtwdemo_roll';
   load_system(model);
   % Configure the Solver
   configSet = getActiveConfigSet(model);
   set_param(configSet,'Solver','FixedStepDiscrete');
   set_param(configSet, 'SystemTargetFile', 'ert.tlc');
   set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true)
   ```
2. Create temporary folders for code generation and generate code. Specify the option ModelReferenceCoderTarget. See slbuild.

   ```
   [TEMPDIR, CGDIR] = rtwdemodir();
   slbuild(model,'ModelReferenceCoderTarget');
   ```

   The generated code is stored in the folder slprj
3. To package the code that is generated as a model reference, use the function packNGo. Locate the file buildinfo.mat in <*working folder*>/slprj/ert/rtwdemo_counter and use the full path to it as the input to packNGo. This command generates an archive containing the generated code and the object buildinfo.mat. See packNGo.

   ```
   % Locate buildinfo and generate code archive
   buildinfo = fullfile(pwd,'slprj','ert',model,'buildinfo.mat');
   packNGo(buildinfo)
   ```
4. To specify the Polyspace analysis options, use a pslinkoptions object. Create this object by using the function pslinkoptions. To run a Bug Finder analysis, set psOpt.VerificationMode to 'BugFinder'.

   ```
   % Create a Polyspace options object from the model.
   psOpts = pslinkoptions(model);

   % Set properties that define the Polyspace analysis.
   psOpts.VerificationMode = 'BugFinder';
   ```

```
psOpts.InputRangeMode = 'DesignMinMax';
```

5 Start a Polyspace analysis. To specify Polyspace analysis options, set the object `psOpt` as the optional second argument in `pslinkrunCrossRelease`. To analyze the code as a model reference, set the optional third argument `asModelRef` to true.

```
% Run Polyspace analysis
[~,resultsFolder] = pslinkrunCrossRelease(model,psOpt,true);
bdclose(model);
```

The character vector `resultsFolder` contains the full path to the results folder.

### Specify Polyspace Analysis Options While Analyzing Code Generated by Using an Earlier Simulink Release

1 Open the Simulink model `rtwdemo_roll` and configure the model for code generation.

```
% Load the model
model = 'rtwdemo_roll';
load_system(model);
% Configure the Solver
configSet = getActiveConfigSet(model);
set_param(configSet,'Solver','FixedStepDiscrete');
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

2 To enable packing the generated code in an archive, set the option `PackageGeneratedCodeAndArtifacts` to `true`.

```
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true)
```

3 Create temporary folders for code generation and generate code.

```
[TEMPDIR, CGDIR] = rtwdemodir();
slbuild(model);
```

4 To specify the model configuration for the Polyspace analysis, use a `pslinkoptions` object. Create this object by using the function `pslinkoptions`. To run a Bug Finder analysis, set `psOpt.VerificationMode` to `'BugFinder'`.

```
% Create a Polyspace options object from the model.
psOpts = pslinkoptions(model);

% Set properties that define the Polyspace analysis.
psOpts.VerificationMode = 'BugFinder';
psOpts.InputRangeMode = 'DesignMinMax';
```

5 To specify Polyspace analysis options, create an options file. An options file is a text file that contains Polyspace options in a flat list, one line for each option. For instance, to enable all Bug Finder checkers and AUTOSAR C++14 coding rules, create a text file named *`OptionFile.txt`*. In the text file, enter:

```
-checkers all
-autosarcpp14 all
```

Save the options file. You can save the preceding options in an options file named `Options.txt` in the default work folder.

See “Complete List of Polyspace Bug Finder Analysis Engine Options”.

6 Start a Polyspace analysis.

- To specify the model configurations for the Polyspace analysis run, set the object psOpt as the optional second argument in pslinkrunCrossRelease.
- Because the code is generated as standalone code, set the third argument asModelRef to false.
- To specify the Polyspace analysis options, specify the relative path to the created options file as the fourth argument.

```
%  Locate options file
optionsPath = fullfile(userpath,'Options.txt');
% Run Polyspace analysis
[~,resultsFolder] = pslinkrunCrossRelease(model,psOpts,false,optionsPath);
bdclose(model);
```

The character vector resultsFolder contains the full path to the results folder.

## Input Arguments

### ModelOrSubsystem — Target of the analysis

bdroot (default) | model or system name

Target of the analysis specified as a character vector with the model or system in single quotes. The default value is the system returned by bdroot.

Example: resultsDir = pslinkrunCrossRelease('rtwdemo_roll') where rtwdemo_roll is the name of a model.

Data Types: char

### psOpt — Options object

configuration options associated with ModelOrSubsystem (default) | configuration object created by using pslinkoptions

Specifies the model configuration for the Polyspace analysis by using a pslinkoptions object. You can modify certain analysis options by modifying psOpt, which is an object where individual fields represent model configuration options. For a full list of options that you can modify, see this table.

**Polyspace Configuration Parameters Supported by `pslinkrunCrossRelease`**

| Property | Description | Value | Description |
|---|---|---|---|
| EnableAdditionalFileList | Enable an additional file list to be analyzed, specified as `true` or `false`. Use with the `AdditionalFileList` option. | `true` | Polyspace verifies additional files specified in the `AdditionalFileList` option. |
| | | `false` (default) | Polyspace does not verify additional files. |
| AdditionalFileList | List of additional files to be analyzed, specified as a cell array of files. To add these files to the analysis, use the `EnableAdditionalFileList` option. | cell array | Polyspace considers the listed files for verification. |
| VerificationMode | Polyspace analysis mode specified as `'BugFinder'`, for a Bug Finder analysis, or `'CodeProver'`, for a Code Prover verification. | 'BugFinder' | Polyspace runs a Bug Finder analysis. |
| | | 'CodeProver' (default) | Polyspace runs a Code Prover analysis. |
| InputRangeMode | Specifies the range of the input variables. | `'DesignMinMax'` (default) | Polyspace uses the input range defined in the workspace or a block. |
| | | `'Fullrange'` | Polyspace uses full range inputs. |
| ParamRangeMode | Specifies the range of the constant parameters. | `'DesignMinmax'` | Polyspace uses the constant parameter range defined in the workspace or in a block. |
| | | `'None'` (default) | Polyspace uses the value of parameters specified in the code. |
| OutputRangeMode | Specifies the output assertions. | `'DesignMinMax'` | Polyspace applies assertions to outputs by using a range defined in a block or the workspace. |
| | | `'None'` (default) | Polyspace does not apply assertions to the output variables. |

| Property | Description | Value | Description |
|---|---|---|---|
| ModelRefVerifDepth | Specify the depth for analyzing the models that are referenced by the current model. | `'Current model Only'` (default) | Polyspace analyzes only the top model without analyzing the referenced models. Use this option when you refer to models that do not need to be analyzed, such as library models. |
| | | `'1'\|'2'\|'3'` | Polyspace analyzes referenced models up to the specified depth in the reference hierarchy. To analyze the models that are referenced by the top model, specify the property `ModelRefVerifDepth` as `'1'`. To analyze models that are referenced by the first level of references, specify this property as `'2'`. |
| | | `'All'` | Polyspace verifies all referenced models. |
| ModelRefByModelRefVerif | Specify whether you want to analyze all referenced models together or individually. | `true` | Polyspace analyzes the top model and the referenced models together. Use this option to check for integration or scaling issues. |
| | | `false` (default) | Polyspace analyzes the top model and the referenced models individually. |
| AutoStubLUT | Specifies how lookup tables are used. | `true` (default) | Polyspace stubs the lookup tables and verifies the model without analyzing the lookup table code. |
| | | `false` | Polyspace includes the lookup table code in the analysis. |

| Property | Description | Value | Description |
|---|---|---|---|
| CheckConfigBeforeAnalysis | Specifies the level of configuration checking done before the Polyspace analysis starts. | 'Off' | Polyspace checks only for errors. The analysis stops if errors are found. |
| | | 'OnWarn' (default) | Polyspace stops the analysis when errors are found and displays a message when warnings are found. |
| | | 'OnHalt' | Polyspace stops the analysis when either errors or warnings are found. |

Example: `pslinkrunCrossRelease('rtwdemo_roll', psOpt)`, where `psOpt` is an options object created by calling `pslinkoptions`

**`asModelRef` — Indicator for model reference analysis**
`false` (default) | `true`

Indicator for model reference analysis, specified as `true` or `false`.

- If `asModelRef` is `false` (default), the function generates options files so that Polyspace analyzes the generated code as standalone code.
- If `asModelRef` is `true`, the function generates options files so that Polyspace analyzes the generated code as model reference code.

Example: `pslinkrunCrossRelease('rtwdemo_roll', psOpt,true)`

Data Types: `logical`

**`OptionsFile` — Relative path to a Polyspace options file**
default Polyspace analysis options (default) | relative path to a custom options file

Relative path to a text file that contains a list of Polyspace analysis options. The options file must have each option in a separate line.

Example: `pslinkrunCrossRelease('rtwdemo_roll', psOpt,true,'OptionsFile.txt')`

Data Types: `char`

## Output Arguments

**`polyspaceFolder` — Folder containing Polyspace project and results**
character vector

Name of the folder containing Polyspace projects and results, specified as a character vector. The default value of this variable is `results_$ModelName$`.

To change this value, see “Output folder” on page 6-16.

**`resultsFolder` — Full path to subfolder containing Polyspace results**
character vector

Full path to subfolder containing Polyspace results, specified as a character vector.

The folder `results_$ModelName$` contains your Polyspace project and a subfolder `$ModelName$` containing the analysis results. This variable provides the full path to the subfolder.

To change the parent folder `results_$ModelName$`, see “Output folder” on page 6-16.

# Version History

**Introduced in R2021a**

## See Also

`slbuild` | `pslinkoptions` | `polyspacePackNGo`

### Topics

“Run Polyspace on Code Generated by Using Previous Releases of Simulink”
“Run Polyspace Analysis on Generated Code by Using Packaged Options Files”
“Run Polyspace Analysis by Using MATLAB Scripts”
“Integrate Polyspace Server Products with MATLAB”
pslinkoptions Properties

# pslinkrun

Run Polyspace analysis on model, system, or S-Function

## Syntax

```
[polyspaceFolder, resultsFolder] = pslinkrun
[polyspaceFolder, resultsFolder]= pslinkrun(target)
[polyspaceFolder, resultsFolder] = pslinkrun('-slcc',target)
[polyspaceFolder, resultsFolder] = pslinkrun(target, opts)
[polyspaceFolder, resultsFolder] = pslinkrun('-slcc', target, opts)
[polyspaceFolder, resultsFolder] = pslinkrun(target, opts, asModelRef)
[polyspaceFolder, resultsFolder] = pslinkrun('-codegenfolder', codegenFolder,
opts)
```

## Description

`[polyspaceFolder, resultsFolder] = pslinkrun` analyzes code generated from the current system using the configuration options associated with the current system. It returns the location of the results folder. The current system is the system returned by the command `bdroot`.

`[polyspaceFolder, resultsFolder]= pslinkrun(target)` analyzes `target` with the configuration options associated with the model containing `target`. Before you run an analysis, you must:

- Generate code for models and subsystems.
- Compile S-Functions.

`[polyspaceFolder, resultsFolder] = pslinkrun('-slcc',target)` runs Polyspace on C/C++ custom code included in C Caller blocks and Stateflow charts in the model.

`[polyspaceFolder, resultsFolder] = pslinkrun(target, opts)` analyzes `target` using the configuration options specified in the object `opts`. It returns the location of the results folder.

`[polyspaceFolder, resultsFolder] = pslinkrun('-slcc', target, opts)` runs Polyspace on C/C++ custom code included in C Caller blocks and Stateflow charts in the model. The analysis uses the configuration options specified in the object `opts`.

`[polyspaceFolder, resultsFolder] = pslinkrun(target, opts, asModelRef)` uses `asModelRef` to specify which type of generated code to analyze—standalone code or model reference code. This option is useful when you want to analyze code that is generated as model reference. Code that is generated as model reference is intended to be called or used in other models or code.

`[polyspaceFolder, resultsFolder] = pslinkrun('-codegenfolder', codegenFolder, opts)` runs Polyspace on C/C++ code generated from MATLAB code and stored in `codegenFolder`.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

## Examples

### Analyze Generated Code

Use a Simulink model to generate code, set configuration options, and then run an analysis from the command line.

```
% Generate code from the model WhereAreTheErrors.
model = 'WhereAreTheErrors';
load_system(model);
slbuild(model);

% Create a Polyspace options object from the model.
opts = pslinkoptions(model);

% Set properties that define the Polyspace analysis.
opts.VerificationMode = 'CodeProver';
opts.VerificationSettings = 'PrjConfigAndMisraC2012';

% Run Polyspace using the options object.
[polyspaceFolder, resultsFolder] = pslinkrun(model,opts);
bdclose(model);
```

The results and the corresponding Polyspace project are saved to the `results_WhereAreTheErrors` folder, listed in the `polyspaceFolder` variable. The full path to the results folder is in the `resultsFolder` variable.

### Analyze Referenced Model Code

Use a Simulink model to generate model reference code, set configuration options, and then run an analysis from the command line.

```
% Generate code from the model WhereAreTheErrors.
% Treat WhereAreTheErrors as if referenced by another model.
model = 'WhereAreTheErrors';
load_system(model);
slbuild(model,'ModelReferenceCoderTarget');

% Create a Polyspace options object from the model.
opts = pslinkoptions(model);

% Set properties that define the Polyspace analysis.
opts.VerificationMode = 'CodeProver';
opts.VerificationSettings = 'PrjConfigAndMisraC2012';

% Run Polyspace with the options object.
[polyspaceFolder, resultsFolder] = pslinkrun(model,opts,true);
bdclose(model);
```

The results and corresponding Polyspace project are saved to the `results_mr_WhereAreTheErrors` folder, listed in the `polyspaceFolder` variable. The full path to the results folder is in the `resultsFolder` variable.

### Reuse Analysis Options for Multiple Models

This example shows how to reuse a subset of options for Polyspace analysis of multiple models. Create a generic options object and specify properties that describe the common options. Associate the generic options object with a model-specific options object. Optionally, set some model-specific options and run the Polyspace analysis.

```
% Generate code from the model WhereAreTheErrors.
model = 'psdemo_model_link_sl';
load_system(model);
slbuild(model);

% Create a generic options object to use for multiple model analyses.
opts = polyspace.ModelLinkOptions();
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';
opts.MergedReporting.ReportOutputFormat = 'PDF';
opts.MergedReporting.EnableReportGeneration = true;

% Create a model-specific options object.
mlopts = pslinkoptions(model);

% Create a project from the generic options object.
% Associate the project with the model-specific options object.
prjfile = opts.generateProject('model_link_opts');
mlopts.EnablePrjConfigFile = true;
mlopts.PrjConfigFile = prjfile;
mlopts.VerificationMode = 'BugFinder';

% Run Polyspace with the model-specific options object.
[polyspaceFolder, resultsFolder] = pslinkrun(model,mlopts);
bdclose(model);
```

After the analysis completes, results open automatically in the Polyspace interface.

### Analyze C/C++ Code Generated from MATLAB Code

This example shows how to analyze C/C++ code generated from MATLAB code.

```
% Generate code
codeName = 'average_filter';
matlabFileName = fullfile(polyspaceroot, 'help',...
    'toolbox','codeprover','examples','matlab_coder','averaging_filter.m');
codegenFolder = fullfile(pwd, 'codegenFolder');
codegen(matlabFileName, '-config:lib', '-c', '-args', ...
   {zeros(1,100,'double')}, '-d', codegenFolder);

% Configure Polyspace analysis
opts = pslinkoptions('ec');
```

```
opts.ResultDir = ['results_',codeName];
opts.OpenProjectManager = 1;

% Run Polyspace
[polyspaceFolder, resultsFolder] = pslinkrun('-codegenfolder', codegenFolder, opts);
```

After the analysis completes, results open automatically in the Polyspace interface.

## Input Arguments

### target — Target of the analysis

bdroot (default) | model or system name | path to S-Function block

Target of the analysis specified as a character vector, with the model, system, or S-function in single quotes. The default value is the system returned by bdroot.

If you analyze custom code in C Caller blocks and Stateflow charts using pslinkrun('-slcc',...), the argument target cannot be an S-Function block.

Example: [polyspaceFolder, resultsFolder] = pslinkrun('demo') where demo is the name of a model.

Example: [polyspaceFolder, resultsFolder] = pslinkrun('path/to/sfunction')

Data Types: char

### opts — Configuration options

configuration options associated with target (default) | object created by pslinkoptions

Specify configuration options of target, specified as a Polyspace options object. The function pslinkoptions creates such an options object. You can customize the options object by changing the pslinkoption properties.

Example: pslinkrun('demo', opts_demo) where demo is the name of a model and opts_demo is an options object.

### asModelRef — Indicator for model reference analysis

false (default) | true

Indicator for model reference analysis, specified as true or false.

- If asModelRef is false (default), Polyspace analyzes the generated code as stand-alone code. This option is equivalent to choosing **Verify Code Generated For** > **Model** in the Simulink Polyspace options.
- If asModelRef is true, Polyspace analyzes the generated code as a model reference code. This option is equivalent to choosing **Verify Code Generated For** > **Referenced Model** in the Simulink Polyspace options. Specifying model reference code indicates that Polyspace must look for the generated code in a different location from the location for standalone code.

Data Types: logical

### codegenFolder — Folder containing generated C/C++ code

character vector

Folder containing C/C++ code generated from MATLAB code, specified as a character vector. You specify this folder with the codegen command using the flag -d.

## Output Arguments

**`polyspaceFolder` — Folder containing Polyspace project and results**
character vector

Name of the folder containing Polyspace project and results, specified as a character vector. The default value of this variable is `results_$ModelName$`.

To change this value, see “Output folder” on page 6-16.

**`resultsFolder` — Full path to subfolder containing Polyspace results**
character vector

Full path to subfolder containing Polyspace results, specified as a character vector.

The folder `results_$ModelName$` contains your Polyspace project and a subfolder `$ModelName$` with the analysis results. This variable gives you the full path to the subfolder. You can use this path with a `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object.

To change the parent folder `results_$ModelName$`, see “Output folder” on page 6-16.

# Version History

**Introduced in R2012a**

## See Also

`pslinkfun` | `pslinkoptions` | pslinkoptions

### Topics

“Run Polyspace Analysis on Code Generated from Simulink Model”
“Run Polyspace Analysis on S-Function Code”
“Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts”
“Recommended Model Configuration Parameters for Polyspace Analysis”

# polyspaceBugFinder

Run Polyspace Bug Finder analysis from MATLAB

**Note** For easier scripting, run Polyspace® analysis using a `polyspace.Project` object.

## Syntax

```
status = polyspaceBugFinder
status = polyspaceBugFinder(projectFile)

status = polyspaceBugFinder(optsObject)
status = polyspaceBugFinder(projectFile, '-nodesktop')

status = polyspaceBugFinder(resultsFile)
status = polyspaceBugFinder('-results-dir',resultsFolder)

status = polyspaceBugFinder('-help')

status = polyspaceBugFinder('-sources',sourceFiles)
status = polyspaceBugFinder('-sources',sourceFiles,Name,Value)
```

## Description

`status = polyspaceBugFinder` opens Polyspace Bug Finder.

`status = polyspaceBugFinder(projectFile)` opens a Polyspace project file in Polyspace Bug Finder.

`status = polyspaceBugFinder(optsObject)` runs an analysis on the Polyspace options object in MATLAB.

`status = polyspaceBugFinder(projectFile, '-nodesktop')` runs an analysis on the Polyspace project file in MATLAB.

`status = polyspaceBugFinder(resultsFile)` opens a Polyspace results file in Polyspace Bug Finder.

`status = polyspaceBugFinder('-results-dir',resultsFolder)` opens a Polyspace results file from `resultsFolder` in Polyspace Bug Finder.

`status = polyspaceBugFinder('-help')` displays options that can be supplied to the `polyspaceBugFinder` command to run a Polyspace Bug Finder analysis.

`status = polyspaceBugFinder('-sources',sourceFiles)` runs a Polyspace Bug Finder analysis on the source files specified in `sourceFiles`.

`status = polyspaceBugFinder('-sources',sourceFiles,Name,Value)` runs a Polyspace Bug Finder analysis on the source files with additional options specified by one or more `Name,Value` pair arguments.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

## Examples

### Open Polyspace Projects from MATLAB

This example shows how to open a Polyspace project file with extension `.psprj` from MATLAB. In this example, you open the project file `Bug_Finder_Example.psprj` from the folder *`polyspaceroot`*`\polyspace\examples\cxx\Bug_Finder_Example`.

Open the project `Bug_Finder_Example.psprj` in the Polyspace interface.

```
prjFile = fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', ...
         'Bug_Finder_Example', 'Bug_Finder_Example.psprj');
polyspaceBugFinder(prjFile);
```

### Open Polyspace Results from MATLAB

This example shows how to open a Polyspace results file from MATLAB. In this example, you open the results file from the folder *`polyspaceroot`*`\polyspace\examples\cxx\Bug_Finder_Example\Module_1\BF_Result`.

Open the results of `resFolder`.

```
resFolder = fullfile(polyspaceroot, 'polyspace', 'examples',  ...
         'cxx', 'Bug_Finder_Example', 'Module_1', 'BF_Result');
polyspaceBugFinder('-results-dir',resFolder)
```

### Run Polyspace Analysis with Options Object

This example shows how to run a Polyspace analysis from the MATLAB command-line using objects.

Create an options object and add the source file and include folder to the properties.

```
opts = polyspace.BugFinderOptions;
opts.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
opts.EnvironmentSettings.IncludeFolders = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Bug_Finder_Example', 'sources')};
opts.ResultsDir = fullfile(pwd,'results');
```

Run the analysis and view the results.

```
polyspaceBugFinder(opts);
polyspaceBugFinder('-results-dir',opts.ResultsDir)
```

### Run Polyspace Analysis from MATLAB with DOS/UNIX Options

This example shows how to run a Polyspace analysis in MATLAB using DOS/UNIX-style options.

Run the analysis and open the results.

```
sourceFiles = fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c');
includeFolders = fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Bug_Finder_Example', 'sources');
resultsDir = fullfile(pwd,'results');
polyspaceBugFinder('-sources',sourceFiles, ...
             '-I',includeFolders, ...
             '-results-dir',resultsDir);
```

To view the results, enter:

```
polyspaceBugFinder('-results-dir',resultsDir);
```

### Run Polyspace Analysis with Coding Rules Checking

This example shows two different ways to customize an analysis in MATLAB. You can customize as many additional options as you want by changing properties in an options object or by using Name-Value pairs. Here you specify checking of MISRA C 2012 coding rules.

Create variables to save the source file path and results folder path. You can use these variables for either analysis method.

```
sourceFileName = fullfile(polyspaceroot, 'polyspace','examples', 'cxx', ...
    'Bug_Finder_Example','sources','dataflow.c');
resFolder1 = fullfile('Polyspace_Results_1');
resFolder2 = fullfile('Polyspace_Results_2');
```

Analyze coding rules with an options object.

```
opts = polyspace.BugFinderOptions();
opts.Sources = {sourceFileName};
opts.ResultsDir = resFolder1;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
polyspaceBugFinder(opts);
polyspaceBugFinder('-results-dir',resFolder1);
```

Analyze coding rules with DOS/UNIX options.

```
polyspaceBugFinder('-sources',sourceFileName,'-results-dir',resFolder2,...
    '-misra3','all');
polyspaceBugFinder('-results-dir',resFolder2);
```

## Input Arguments

### `optsObject` — Polyspace options object name

object handle

Polyspace options object name, specified as the object handle.

To create an options object, use one of the Polyspace options classes.

Example: `opts`

### `projectFile` — Name of `.psprj` file

character vector

Name of project file with extension `.psprj`, specified as a character vector.

If the file is not in the current folder, `projectFile` must include a full or relative path.

Example: `'C:\Polyspace_Projects\myProject.psprj'`

Data Types: `char`

### `resultsFile` — Name of `.psbf` file

character vector

Name of results file with extension `.psbf`, specified as a character vector.

If the file is not in the current folder, `resultsFile` must include a full or relative path.

Example: `'myResults.psbf'`

Data Types: `char`

### `resultsFolder` — Name of result folder

character vector

Name of result folder, specified as a character vector. The folder must contain the results file with extension `.psbf`. If the results file resides in a subfolder of the specified folder, this command does not open the results file.

If the folder is not in the current folder, `resultsFolder` must include a full or relative path.

Example: `'C:\Polyspace\Results\'`

Data Types: `char`

### `sourceFiles` — Comma-separated names of C or C++ files

character vector

Comma-separated C or C++ source file names, specified as a single character vector.

If the files are not in the current folder, `sourceFiles` must include a full or relative path.

Example: `'myFile.c'`, `'C:\mySources\myFile1.c,C:\mySources\myFile2.c'`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose* `Name` *in quotes.*

Example: `'-target','i386','-compiler','gnu4.6'` specifies that the source code is intended for a i386 target and contains non-ANSI C syntax for GCC 4.6.

For option names and values, see the **Command-Line Information** section in “Complete List of Polyspace Bug Finder Analysis Engine Options”.

## Output Arguments

**`status` — Status indicating whether the Polyspace Bug Finder analysis completed successfully or not**
`true` | `false`

If the Polyspace Bug Finder analysis completes without error, `status` is `false`. Otherwise, it is `true`.

The analysis might fail for multiple reasons, including:

- You provided a source file, project file, or results file that does not exist.
- You specified an invalid path.
- One of your files did not compile.

# Version History

**Introduced in R2013b**

## See Also

`polyspace.BugFinderOptions` | `polyspace.ModelLinkBugFinderOptions`

**Topics**

"Run Polyspace Analysis by Using MATLAB Scripts"

# polyspaceBugFinderServer

Run analysis with Polyspace Bug Finder Server using MATLAB scripts

**Note** For easier scripting, run Polyspace® analysis using a `polyspace.Project` object.

## Syntax

`polyspaceBugFinderServer(optsObject)`

`polyspaceBugFinderServer('-help')`

`polyspaceBugFinderServer('-sources',sourceFiles)`
`polyspaceBugFinderServer('-sources',sourceFiles,Name,Value)`

## Description

`polyspaceBugFinderServer(optsObject)` runs an analysis on the Polyspace options object in MATLAB.

`polyspaceBugFinderServer('-help')` displays options that can be supplied to the `polyspaceBugFinderServer` command to run an analysis with Polyspace Bug Finder Server.

`polyspaceBugFinderServer('-sources',sourceFiles)` runs an analysis with Polyspace Bug Finder Server on the source files specified in `sourceFiles`.

`polyspaceBugFinderServer('-sources',sourceFiles,Name,Value)` runs an analysis with Polyspace Bug Finder Server on the source files with additional options specified by one or more `Name,Value` pair arguments.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace Server Products with MATLAB”.

## Examples

### Run Polyspace Analysis with Options Object

This example shows how to run a Polyspace analysis from the MATLAB command-line. For this example:

- Use the source file `numerical.c` located in the directory *`polyspaceroot`*`/polyspace/examples/cxx/BugFinder_example/sources`.
- Include the headers located in the same directory.

Create an options object and add the source file and include folder to the properties.

```
opts = polyspace.BugFinderOptions;
opts.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};;
opts.EnvironmentSettings.IncludeFolders = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Bug_Finder_Example', 'sources')};
opts.ResultsDir = 'C:\Polyspace_Results';
```

Polyspace runs on the file `C:\Polyspace_Sources\source.c` and stores the result in `C:\Polyspace_Results`.

Run the analysis with Polyspace Bug Finder Server.

```
polyspaceBugFinderServer(opts);
```

### Run Polyspace Analysis from MATLAB with DOS/UNIX Options

This example shows how to run a Polyspace analysis in MATLAB by using DOS/UNIX Options. For this example:

- Use the source file `numerical.c` located in the directory *`polyspaceroot`*`/polyspace/examples/cxx/BugFinder_example/sources`.
- Include the headers located in the same directory.

Define the location of source and include files.

```
src = fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c');
inc = fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Bug_Finder_Example', 'sources');
res = fullfile(pwd,'results');
```

To analyze `numerical.c`, run the following command.

```
polyspaceBugFinderServer('-sources',src, ...
    '-I',inc, ...
    '-results-dir',res)
```

### Run Polyspace Analysis with Coding Rules Checking

This example shows two different ways to customize an analysis in MATLAB. You can customize as many additional options as you want by changing properties in an options object or by using Name-Value pairs. Here you specify checking of MISRA C 2012 coding rules.

Create variables to save the source file path and results folder path. You can use these variables for either analysis method.

```
sourceFileName = fullfile(polyspaceroot, 'polyspace','examples', 'cxx', ...
    'Bug_Finder_Example','sources','dataflow.c');
resFolder1 = fullfile('Polyspace_Results_1');
resFolder2 = fullfile('Polyspace_Results_2');
```

Analyze coding rules with an options object.

```
opts = polyspace.BugFinderOptions();
opts.Sources = {sourceFileName};
opts.ResultsDir = resFolder1;
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';
polyspaceBugFinderServer(opts);
```

Analyze coding rules with DOS/UNIX options.

```
polyspaceBugFinderServer('-sources',sourceFileName,'-results-dir',resFolder2,...
    '-misra3','all');
```

## Input Arguments

### `optsObject` — Polyspace options object name

object handle

Polyspace options object name, specified as the object handle.

To create an options object, use one of the Polyspace options classes `polyspace.Options` or `polyspace.Project`.

Example: `opts`

### `sourceFiles` — Comma-separated names of C or C++ files

character vector

Comma-separated C or C++ source file names, specified as a single character vector.

If the files are not in the current folder, `sourceFiles` must include a full or relative path.

Example: `'myFile.c'`, `'C:\mySources\myFile1.c,C:\mySources\myFile2.c'`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose* `Name` *in quotes.*

Example: `'-target','i386','-compiler','gnu4.6'` specifies that the source code is intended for a i386 target and contains non-ANSI C syntax for GCC 4.6.

For option names and values, see the **Command-Line Information** section in “Complete List of Polyspace Bug Finder Analysis Engine Options”.

# Version History

**Introduced in R2019a**

## See Also

`polyspace.Project`

### Topics

“Integrate Polyspace Server Products with MATLAB”

# polyspaceConfigure

Create Polyspace project from your build system at the MATLAB command line

## Syntax

```
polyspaceConfigure buildCommand
```

```
polyspaceConfigure -option value buildCommand
```

## Description

`polyspaceConfigure buildCommand` traces your build system and creates a Polyspace project with information gathered from your build system. You can run an analysis on a Polyspace project only in the user interface of the Polyspace desktop products.

`polyspaceConfigure -option value buildCommand` traces your build system and uses `-option value` to modify the default operation of `polyspaceConfigure`. Specify the modifiers before `buildCommand`, otherwise they are considered as options in the build command itself.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” or “Integrate Polyspace Server Products with MATLAB”.

## Examples

### Create Polyspace Project from Makefile

This example shows how to create a Polyspace project if you use the command `make` *`targetName buildOptions`* to build your source code. The example creates a Polyspace project that can be opened only in the user interface of the Polyspace desktop products.

Create a Polyspace project specifying a unique project name. Use the `-B` or `-W` *`makefileName`* option with `make` so that the all prerequisite targets in the makefile are remade.

```
polyspaceConfigure  -prog myProject ...
          make -B targetName buildOptions
```

Open the Polyspace project in the **Project Browser**.

```
polyspaceBugFinder('myProject.psprj')
```

### Create Projects That Have Different Source Files from Same Build Trace

This example shows how to create different Polyspace projects from the same trace of your build system. You can specify which source files to include for each project. The example creates a Polyspace project that can be opened only in the user interface of the Polyspace desktop products.

Trace your build system without creating a Polyspace project by specifying the option `-no-project`. To ensure that all the prerequisite targets in your makefile are remade, use the appropriate `make` build command option, for instance `-B`.

```
polyspaceConfigure -no-project make -B;
```

`polyspace-configure` stores the cache information and the build trace in default locations inside the current folder. To store the cache information and build trace in a different location, specify the options `-cache-path` and `-build-trace`.

Generate Polyspace projects by using the build trace information from the previous step. Specify a project name and use the `-include-sources` or `-exclude-sources` option to select which files to include for each project.

```
polyspaceConfigure -no-build -prog myProject ...
-include-sources "glob_pattern";
```

*`glob_pattern`* is a glob pattern that corresponds to folders or files you filter in or out of your project. To ensure the shell does not expand the glob patterns you pass to `polysapce-configure`, enclose them in double quotes.For more information on the supported syntax for glob patterns, see “Select Files for Polyspace Analysis Using Pattern Matching”.

If you specified the options `-build-trace` and `-cache-path` in the previous step, specify them again.

Delete the trace file and cache folder.

```
rmdir('polyspace_configure_cache', 's');
delete polyspace_configure_built_trace;
```

If you used the options `-build-trace` and `-cache-path`, use the paths and file names from those options.

### Run Command-Line Polyspace Analysis from Makefile

This example shows how to run Polyspace analysis if you use a build command such as `make` *`targetName buildOptions`* to build your source code. In this example, you use `polyspaceConfigure` to trace your build system but do not create a Polyspace project. Instead you create an options file that you can use to run Polyspace analysis from the command-line.

Create a Polyspace options file specifying the `-output-options-file` command. Use the `-B` or `-W` *`makefileName`* option with `make` so that all prerequisite targets in the makefile are remade.

```
polyspaceConfigure -output-options-file ...
         myOptions make -B targetName buildOptions
```

Use the options file that you created to run a Polyspace analysis at the command line:

```
polyspaceBugFinder -options-file myOptions
```

## Input Arguments

### buildCommand — Command for building source code

build command

Build command specified exactly as you use to build your source code.

Example: `make -B`, `make -W *makefileName*`

### -option value — Options for changing default operation of polyspaceConfigure

single option starting with `-`, followed by argument | multiple space-separated option-argument pairs

**Basic Options**

| Option | Argument | Description |
|---|---|---|
| `-prog` | Project name | Project name that appears in the Polyspace user interface. The default is `polyspace`.<br>If you do not use the option `-output-project`, the `-prog` argument also sets the project name.<br>**Example:** `-prog myProject` creates a project that has the name `myProject` in the user interface. If you do not use the option `-output-project`, the project name is also `myProject.psrprj`. |
| `-author` | Author name | Name of project author.<br>**Example:** `-author jsmith` |
| `-output-project` | Path | Name and location of Polyspace project. The default is the file `polyspace.psprj` in the current folder.<br>**Example:** `-output-project ../myProjects/project1` creates a project `project1.psprj` in the folder with the relative path `../myProjects/`. |
| `-output-options-file` | File name | Option to create a Polyspace analysis options file. Use this file for command-line analysis using one of these commands:<br>• `polyspace-bug-finder`<br>• `polyspace-code-prover`<br>• `polyspace-bug-finder-server`<br>• `polyspace-code-prover-server`<br>• `polyspace-bug-finder-access` |

| Option | Argument | Description |
|---|---|---|
| `-allow-build-error` | None | Option to create a Polyspace project even if an error occurs in the build process.<br>If an error occurs, the build trace log shows the following message:<br>`polyspace-configure (polyspaceConfigure)`<br>`   ERROR: build command`<br>`   command_name fail [status=status_value]`<br>*`command_name`* is the build command name that you use and *`status_value`* is the non-zero exit status or error level that indicates which error occurred in your build process.<br>This option is ignored when you use `-compilation-database`. |
| `-allow-overwrite` | None | Option to overwrite a project with the same name, if it exists.<br>By default, `polyspace-configure (polyspaceConfigure)` throws an error if a project with the same name already exists in the output folder. Use this option to overwrite the project. |
| `-no-console-output`<br>`-silent` (default)<br>`-verbose` | None | Option to suppress or display additional messages from running `polyspace-configure (polyspaceConfigure)`.<br>• `-no-console-output` – Suppress all outputs including errors and warnings.<br>• `-silent` (default) – Show only errors and warnings.<br>• `-verbose` – Show all messages.<br>If you specify more than one of these options, the most verbose option is applied.<br>These options are ignored if they are used in combination with `-easy-debug`. |
| `-help` | None | Option to display the full list of `polyspace-configure (polyspaceConfigure)` commands |
| `-debug` | None | Option to store debug information for use by MathWorks technical support.<br>This option has been superseded by the option `-easy-debug`. |
| `-easy-debug` | Path | Option to store debug information for use by MathWorks technical support.<br>After a `polyspace-configure (polyspaceConfigure)` run, the path provided contains a zipped file ending with `pscfg-output.zip`. If the run fails to create a complete Polyspace project or options file, send this zipped file to MathWorks Technical Support for further debugging. The zipped file does not contain source files traced in the build. See also “Errors in Project Creation from Build Systems”. |

### Options to Create Multiple Modules

These options are not compatible with `-compilation-database`.

| Option | Argument | Description |
|---|---|---|
| `-module` | None | Option to create a separate options file for each binary you create in build system. You can only create separate options files for different binaries.<br>Use this option only if your build system uses GNU or Visual C++ compilers.<br>See also “Modularize Polyspace Analysis by Using Build Command”. |
| `-output-options-path` | Path name | Location where generated options files are saved. Use this option together with the option `-module`.<br>The options files are named after the binaries created in the build system. |

**Advanced Options**

| Option | Argument | Description |
|---|---|---|
| `-compilation-database` | Path and file name | Location and name of JSON compilation database (JSON CDB) file. You generate this file from your build system, for instance by using the flag `-DCMAKE_EXPORT_COMPILE_COMMANDS=1` with `cmake`. The file contains compiler calls for all the translation units in you projects. For more information, see JSON Compilation Database. `polyspace-configure` uses the content of this file to get information about your build system. The extracted compiler paths in the JSON CDB must be accessible from the path where you run `polyspace-configure`.<br>You do not specify a build command when you use this option.<br>These build systems and compilers support the generation of a JSON CDB:<br>• CMake<br>• Bazel<br>• Clang<br>• Ninja<br>• Qbs<br>• waf<br>This option is not compatible with `-no-project` and with the options to create multiple modules.<br>The cache control options, `-allow-build-error`, and `-no-build` are ignored when you use this option. |
| `-compiler-config` | Path and file name | Location and name of compiler configuration file.<br>The file must be in a specific format. For guidance, see the existing configuration files in *`polyspaceroot`*`\polyspace\configure\compiler_configuration\`. For information on the contents of the file, see “Create Polyspace Projects from Build Systems That Use Unsupported Compilers”.<br>**Example:** `-compiler-configuration myCompiler.xml` |

| Option | Argument | Description |
|---|---|---|
| `-no-project` | None | Option to trace your build system without creating a Polyspace project and save the build trace information.<br>Use this option to save your build trace information for a later run of `polyspace-configure (polyspaceConfigure)` with the `-no-build` option.<br>This option is not compatible with `-compilation-database`. |
| `-no-build` | None | Option to create a Polyspace project using previously saved build trace information.<br>To use this option, you must have the build trace information saved from an earlier run of `polyspace-configure (polyspaceConfigure)` with the `-no-project` option.<br>If you use this option, you do not need to specify the `buildCommand` argument.<br>This option is ignored when you use `-compilation-database`. |
| `-no-sources` | None | Option to create a Polyspace options file that does not contain the source file specifications.<br>Use this option when you intend to specify the source files by other means. For instance, you can use this option when:<br>• Running Polyspace on AUTOSAR-specific code.<br>You want to create an options file that traces your build command for the compiler options:<br>`-output-options-file options.txt -no-sources`<br>You later append this options file when extracting source file names from ARXML specifications and running the subsequent Code Prover analysis with `polyspace-autosar`<br>`-extra-options-file options.txt`<br>See also “Run Polyspace on AUTOSAR Code Using Build Command” (Polyspace Code Prover).<br>• Running Polyspace in Eclipse.<br>Your source files are already specified in your Eclipse project. When running a Polyspace analysis, you want to specify an options file that has the compilation options only. |

| Option | Argument | Description |
|---|---|---|
| `-extra-project-options` | Options to use for subsequent Polyspace analysis. For instance, `"-stubbed-pointers-are-unsafe"`. | Options that are used for subsequent Polyspace analysis.<br>Once a Polyspace project is created, you can change some of the default options in the project. Alternatively, you can pass these options when tracing your build command. The flag `-extra-project-options` allows you to pass additional options.<br>Specify multiple options in a space separated list, for instance `"-allow-negative-operand-in-shift -stubbed-pointers-are-unsafe"`.<br>Suppose you have to set the option `-stubbed-pointers-are-unsafe` for every Polyspace project created. Instead of opening each project and setting the option, you can use this flag when creating the Polyspace project:<br>`-extra-project-options "-stubbed-pointers-are-unsafe"`<br>For the list of options available, see:<br>• "Complete List of Polyspace Bug Finder Analysis Engine Options"<br>• "Complete List of Polyspace Code Prover Analysis Options" (Polyspace Code Prover)<br>If you are creating an options file instead of a Polyspace project from your build command, do not use this flag. Instead, add the extra analysis options manually in the generated options file, in a separate options file, or at the command-line when you start the analysis. |
| `-tmp-path` | Path | Location of folder where temporary files are stored. |
| `-build-trace` | Path and file name | Location and name of file where build information is stored. The default is `./polyspace_configure_build_trace.log`.<br>**Example:** `-build-trace ../build_info/trace.log` |
| `-log` | Path and file name | Location and name of log file where the output of the `polyspace-configure` command is stored. The use of this option does not suppress the console output. |
| `-include-sources`<br>`-exclude-sources` | Glob pattern | Option to specify which source files `polyspace-configure` (`polyspaceConfigure`) includes in, or excludes from, the generated project. You can combine both options together.<br>A source file is included if the file path matches the glob pattern that you pass to `-include-sources`.<br>A source file is excluded if the file path matches the glob pattern that you pass to `-exclude-sources`. |

| Option | Argument | Description |
|---|---|---|
| `-print-included-sources`<br>`-print-excluded-sources` | None | Option to print the list of source files that `polyspace-configure` `(polyspaceConfigure)` includes in, or excludes from, the generated project. You can combine both options together. The output displays the full path of each file on a separate line.<br>Use this option to troubleshoot the glob patterns that you pass to `-include-sources` or `-exclude-sources`. You can see which files match the pattern that you pass to `-include-sources` or `-exclude-sources`. |
| `-compiler-cache-path` | Folder path | Specify a folder path where `polyspace-configure` looks for or stores the compiler cache files. If the folder does not exist, `polyspace-configure` creates it.<br>By default, Polyspace looks for and stores compiler caches under these folder paths:<br>• **Windows**<br>`%appdata%\Mathworks\R20xxY\Polyspace`<br>• **Linux**<br>`~/.matlab/R20xxY/Polyspace`<br>• **Mac**<br>`~/Library/Application Support/MathWorks/MATLAB/R20xxY/Polyspace`<br>*`R20xxY`* is the release version of your Polyspace product, for instance R2020b. |
| `-no-compiler-cache` | None | Use this option if you do not want Polyspace to cache your compiler configuration information or to use an existing cache for your compiler configuration.<br>By default, the first time you run `polyspace-configure` with a particular compiler configuration, Polyspace queries your compiler for the size of fundamental types, compiler macro definitions, and other compiler configuration information then caches this information. Polyspace reuses the cached information in subsequent runs of `polyspace-configure` for builds that use the same compiler configuration. |
| `-reset-compiler-cache-entry` | None | Use this option to query the compiler for the current configuration and to refresh the entry in the cache file that corresponds to this configuration. Other compiler configuration entries in the cache are not updated. |
| `-clear-compiler-cache` | None | Use this option to delete all compiler configurations stored in the cache file.<br>If you also specify a build command or `-compilation-database`, `polyspace-configure` computes and caches the compiler configuration information of the current run, except if you specify `-no-project` or `-no-compiler-cache`. |

| Option | Argument | Description |
|---|---|---|
| -import-macro-definitions | none<br>from-allowlist<br>from-source-tokens<br>from-compiler | Typically, you do not need to specify this option.<br>Polyspace attempts to automatically determine the best strategy to query your compiler for macro definitions in this order of priority:<br>1 from-compiler — Polyspace uses native compiler options, such as gcc -dm -E, to obtain the compiler macro definitions. This strategy does not require Polyspace to trace your build and is available only for compilers that support listing macro definitions.<br>2 from-source-tokens — Polyspace uses every non-keyword token in your source code to query your compiler for macro definitions. This strategy is available only if Polyspace can trace your build. The strategy is not available if you use option -compilation-database.<br>3 from-allowlist — Polyspace uses an internal allow list to query the compiler for macro definitions.<br>If you prefer to specify macro definitions manually, use this option with the none flag and use option Preprocessor definitions (-D) to specify the macro definitions.<br>If the macro import strategy that Polyspace uses is not the one that you expect, try specifying this option manually to troubleshoot the issue. |
| -options-for-sources-delimiter | A single character | Specify an option separator to use when multiple analysis options are associated with one source file using the -options-for-sources option. Typically, the -options-for-sources option uses a semicolon as separator.<br>See also -options-for-sources. |

### Cache Control Options

These options are primarily useful for debugging. Use the options if polyspace-configure (polyspaceConfigure) fails and MathWorks Technical Support asks you to use the option and provide the cached files. Starting R2020a, the option -easy-debug provides an easier way to provide debug information. See “Contact Technical Support About Issues with Running Polyspace”.

These options are ignored when you use -compilation-database.

| Option | Argument | Description |
|---|---|---|
| -no-cache<br>-cache-sources (default)<br>-cache-all-text<br>-cache-all-files | None | Option to perform one of the following:<br>• -no-cache: Not create a cache<br>• -cache-sources: Cache text files temporarily created during build for later use by polyspace-configure (polyspaceConfigure).<br>• -cache-all-text: Cache all text files including sources and headers.<br>• -cache-all-files: Cache all files including binaries.<br>Typically, you cache temporary files created by your build command to debug issues in tracing the command. |

| Option | Argument | Description |
|---|---|---|
| `-cache-path` | Path | Location of folder where cache information is stored.<br>When tracing a Visual Studio build (`devenv.exe`), if you see the error:<br>`path is too long`<br>try using a shorter path for this option to work around the error.<br>**Example:** `-cache-path ../cache` |
| `-keep-cache`<br>`-no-keep-cache` (default) | None | Option to preserve or clean up cache information after `polyspace-configure (polyspaceConfigure)` completes execution.<br>If `polyspace-configure (polyspaceConfigure)` fails, you can provide this cache information to technical support for debugging purposes. |

# Version History

**Introduced in R2013b**

## See Also

### Topics

"Modularize Polyspace Analysis by Using Build Command"
"Create Polyspace Analysis Configuration from Build Command (Makefile)"
"Requirements for Project Creation from Build Systems"
"Create Polyspace Projects from Build Systems That Use Unsupported Compilers"

# polyspaceJobsManager

Manage Polyspace jobs on a MATLAB Parallel Server cluster

## Syntax

```
polyspaceJobsManager('listjobs')
polyspaceJobsManager('cancel','-job',jobNumber)
polyspaceJobsManager('remove','-job',jobNumber)
polyspaceJobsManager('getlog','-job',jobNumber)
polyspaceJobsManager('wait','-job',jobNumber)
polyspaceJobsManager('promote','-job',jobNumber)
polyspaceJobsManager('demote','-job',jobNumber)
```

```
polyspaceJobsManager('download','-job',jobNumber)
polyspaceJobsManager('download','-job',jobNumber,'-results-folder',
resultsFolder)
```

```
polyspaceJobsManager( ___ ,'-scheduler',scheduler)
```

## Description

`polyspaceJobsManager('listjobs')` lists all Polyspace jobs in your cluster.

`polyspaceJobsManager('cancel','-job',jobNumber)` cancels the specified job. The job appears in your queue as cancelled.

`polyspaceJobsManager('remove','-job',jobNumber)` removes the specified job from your cluster.

`polyspaceJobsManager('getlog','-job',jobNumber)` displays the log for the specified job.

`polyspaceJobsManager('wait','-job',jobNumber)` pauses until the specified job is done.

`polyspaceJobsManager('promote','-job',jobNumber)` moves the specified job up in the MATLAB job scheduler queue.

`polyspaceJobsManager('demote','-job',jobNumber)` moves the specified job down in the MATLAB job scheduler queue.

`polyspaceJobsManager('download','-job',jobNumber)` downloads the results from the specified job. The results are downloaded to the folder you specified when starting analysis, using the `-results-dir` on page 3-63 option.

`polyspaceJobsManager('download','-job',jobNumber,'-results-folder', resultsFolder)` downloads the results from the specified job to `resultsFolder`.

`polyspaceJobsManager( ___ ,'-scheduler',scheduler)` performs the specified action on the job scheduler specified. If you do not specify a server with any of the previous syntaxes, Polyspace uses the server stored in your Polyspace preferences.

## Examples

### Manipulate Two Jobs in the Cluster

In this example, use a MATLAB Job Scheduler scheduler to run Polyspace remotely and monitor your jobs through the queue.

Before performing this example, set up a MATLAB Job Scheduler. This example uses the *myMJS@myCompany.com* scheduler. When you perform this example, replace this scheduler with your own cluster name.

Set up your source files.

```
tempDir = fullfile(tempdir, 'psdemo', 'src');
mkdir(tempDir);
demo = fullfile(polyspaceroot,'polyspace','examples','cxx',...
'Bug_Finder_Example','sources');
copyfile(demo,tempDir,'f');
```

Submit two jobs to your scheduler.

If your jobs have not started running, promote the second job to run before the first job.

```
polyspaceJobsManager('promote','-job','20','-scheduler',...
    'myMJS@myCompany.com')
```

Job 20 starts running before job 19.

Cancel job 19.

```
polyspaceJobsManager('cancel','-job','19','-scheduler',...
   'myMJS@myCompany.com')
polyspaceJobsManager('listjobs','-scheduler','myMJS@myCompany.com')
```

Remove job 19.

```
polyspaceJobsManager('remove','-job','19','-scheduler',...
   'myMJS@myCompany.com')
polyspaceJobsManager('listjobs','-scheduler','myMJS@myCompany.com')
```

Get the log for job 20.

```
polyspaceJobsManager('getlog','-job','20','-scheduler',...
   'myMJS@myCompany.com')
```

Download the information from job 20.

```
resFolder3 = fullfile(tempDir, 'res3');
polyspaceJobsManager('download','-job','20','-results-folder', ...
   resFolder3,'-scheduler','myCluster')
```

## Input Arguments

### `jobNumber` — Queued job number

character vector of job number

Number of the queued job that you want to manage, specified as a character vector in single quotes.

Example: `'-job','10'`

### `resultsFolder` — Path to results folder

character vector

Path to results folder specified as a character vector in single quotes. This folder stores the downloaded results files.

Example: `'-results-folder','C:\psdemo\myresults'`

### `scheduler` — job scheduler

head node of your cluster | job scheduler name | cluster profile

Job scheduler for remote verifications specified as one of the following:

- Name of the computer that hosts the head node of your MATLAB Parallel Server cluster (*`NodeHost`*).
- Name of the MATLAB Job Scheduler on the head node host (*`MJSName@NodeHost`*).
- Name of a MATLAB cluster profile (*`ClusterProfile`*).

Example: `'-scheduler','myscheduler@mycompany.com'`

## Version History

**Introduced in R2013b**

## See Also

`polyspaceBugFinder`

**Topics**

"Discover Clusters and Use Cluster Profiles" (Parallel Computing Toolbox)
"Send Polyspace Analysis from Desktop to Remote Servers Using Scripts"

# polyspaceroot

Get Polyspace installation folder

## Syntax

```
polyspaceroot
```

## Description

`polyspaceroot` returns the Polyspace installation folder.

Starting in R2019a, to run MATLAB scripts for Polyspace analysis, you install MATLAB and Polyspace in separate folders and link between them. After installation and linking, to access files in the Polyspace installation folder from MATLAB, use this function. See also “Integrate Polyspace with MATLAB and Simulink” or “Integrate Polyspace Server Products with MATLAB”.

## Examples

### Get Polyspace Installation Folder

To determine the Polyspace installation folder, use the `polyspaceroot` function.

```
polyspaceroot
```

```
C:\Program Files\Polyspace\R2019a
```

With the products, Polyspace Bug Finder Server or Polyspace Code Prover Server, the default installation folder in Windows is:

```
C:\Program Files\Polyspace Server\R2019a
```

### Run Polyspace on Sample Files in Polyspace Installation Folder

To access sample files in the Polyspace installation folder, use the `polyspaceroot` function to get the root of the installation folder. Append subfolders to the root folder path with the `fullfile` function.

Run Bug Finder on the file `numerical.c` in the subfolder `polyspace\examples\cxx\Bug_Finder_Example\sources` of the Polyspace installation folder.

```
proj = polyspace.Project

% Specify sources and includes
sourceFile = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c');
includeFolder = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');

% Configure analysis
proj.Configuration.Sources = {sourceFile};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.EnvironmentSettings.IncludeFolders = {includeFolder};
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = proj.run('bugFinder');
```

## Version History

**Introduced in R2019a**

## See Also

`polyspace.Project`

### Topics

"Run Polyspace Analysis by Using MATLAB Scripts"
"Integrate Polyspace Server Products with MATLAB"

# polyspace_report

Generate reports from Polyspace analysis results

## Syntax

```
polyspace_report('-template', template, '-results-dir', resultsFolder,
options)
polyspace_report('-generate-results-list-file', '-results-dir',
resultsFolder, options)
polyspace_report('-generate-variable-access-file', '-results-dir',
resultsFolder, options)
```

## Description

`polyspace_report('-template', template, '-results-dir', resultsFolder, options)` generates a report using a predefined template specified by `template`. By default, the report is named after the results file in the folder `resultsFolder` and saved in the `Polyspace-Doc` subfolder. You can change the default behavior using additional options.

`polyspace_report('-generate-results-list-file', '-results-dir', resultsFolder, options)` exports the list of Polyspace results to a tab-delimited text file.

`polyspace_report('-generate-variable-access-file', '-results-dir', resultsFolder, options)` exports the list of global variables to a tab-delimited text file.

**Note**

- Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” or “Integrate Polyspace Server Products with MATLAB”.
- You need MATLAB Report Generator™ to use this function.

## Examples

### Generate PDF Report from Results

Generate a PDF report from sample Polyspace Code Prover results.

```
template = fullfile(polyspaceroot,'toolbox','polyspace','psrptgen','templates',...
    'Developer.rpt');
resPath = fullfile(polyspaceroot,'polyspace','examples','cxx','Code_Prover_Example',...
    'Module_1','CP_Result');
polyspace_report('-template', template, '-results-dir', resPath, '-format', 'PDF');
```

## Input Arguments

### `template` — Path to report template file

character vector

Path to report template file, specified as a character vector. To generate multiple reports, specify a comma-separated list of report template paths in the character vector (do not put a space after the commas). The templates are available in `*polyspaceroot*\toolbox\polyspace\psrptgen\templates\` as `.rpt` files. Here, `*polyspaceroot*` is the Polyspace installation folder. For more information on the available templates, see `Bug Finder and Code Prover report (-report-template)`.

Example: `fullfile(polyspaceroot,'toolbox','polyspace','psrptgen','templates', 'Developer.rpt');`

### `resultsFolder` — Folder containing analysis results

character vector

Folder containing analysis results, specified as a character vector. The folder must contain a `.psbf` file containing Polyspace Bug Finder results or a `.pscp` file containing Polyspace Code Prover results.

To generate reports for multiple analyses, specify a comma-separated list of folder paths (do not put a space after the commas).

Example: `'C:\Polyspace_Workspace\My_project\Module_1\results'`

### `options` — Options for generating report

character vector

Options to control report generation, for instance, output format and output name.

Specify each option as a character vector, followed by the option value as a separate character vector. For instance, you can specify the PDF format by using the syntax `polyspace_report(..., '-format','PDF')`.

| Option | Value | Description |
|---|---|---|
| '-format' | 'PDF', 'HTML' or 'WORD' | File format of the report that you generate. By default, the command generates a Word document.<br>To generate reports in multiple formats, specify a comma-separated list of formats. (Do not put a space after the commas). For instance, polyspace_report(..., '-format', 'PDF,HTML').<br>This option is not compatible with -generate-variable-access-file and -generate-results-list-file. |
| '-set-language-english' | | Generate the report in English. Use this option if your display option is set to another language. |
| '-output-name' | Report name, for instance, PolyspaceReport. | Name of the generated report or folder name if you generate multiple reports.<br>The full path to the report is created by appending the name to the current working folder. To store the reports on a different path, specify the full path as value for this option. |

## Version History

**Introduced in R2013b**

# polyspace.Project

Run Polyspace analysis on C and C++ code and read results

## Description

Run a Polyspace analysis on C and C++ source files by using this MATLAB object. To specify source files and customize analysis options, use the `Configuration` property. To run the analysis, use the `run` method. To read results after analysis, use the `Results` property.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” or “Integrate Polyspace Server Products with MATLAB”.

## Creation

`proj = polyspace.Project` creates an object that you can use to configure and run a Polyspace analysis, and then read the analysis results.

### Properties

**`Configuration` — Analysis options**
`polyspace.Options` object

Options for running Polyspace analysis, implemented as a `polyspace.Options` object. The object has properties corresponding to the analysis options. For more information on those properties, see polyspace.Project.Configuration properties.

You can retain the default options or change them in one of these ways:

- Set the source code language to '`C`', '`CPP`', or '`C-CPP`' (default). Some analysis options might not be available depending on the language setting of the object.

  ```
  proj=polyspace.Project;
  proj.Configuration=polyspace.Options('C');
  ```
- Modify the properties directly.

  ```
  proj = polyspace.Project;
  proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
  ```
- Obtain the options from another `polyspace.Project` object.

  ```
  proj1 = polyspace.Project;
  proj1.Configuration.TargetCompiler.Compiler = 'gnu4.9';

  proj2 = proj1;
  ```

  To use common analysis options across multiple projects, follow this approach. For instance, you want to reuse all options and change only the source files.

- Obtain the options from a project created in the user interface of the Polyspace desktop products (`.psprj` file).

  ```
  proj = polyspace.Project;
  projectLocation = fullfile(polyspaceroot, 'polyspace', ...
      'examples', 'cxx', 'Bug_Finder_Example', 'Bug_Finder_Example.psprj')
  proj.Configuration = polyspace.loadProject(projectLocation);
  ```

  To determine the optimal set of options, set your options in the user interface and then import them to a `polyspace.Project` object. In the user interface, you can get tooltip help on options.
- Obtain the options from a Simulink model (applies only to Polyspace desktop products). Before obtaining the options, generate code from the model.

  ```
  modelName = 'rtwdemo_roll';
  load_system(modelName);

  % Set parameters for Embedded Coder target
  set_param(modelName, 'SystemTargetFile', 'ert.tlc');
  set_param(modelName,'Solver','FixedStepDiscrete');
  set_param(modelName,'SupportContinuousTime','on');
  set_param(modelName,'LaunchReport','off');
  set_param(modelName,'InitFltsAndDblsToZero','on');

  if exist(fullfile(pwd,'rtwdemo_roll_ert_rtw'), 'dir') == 0
      slbuild(modelName);
  end

  % Obtain configuration from model
  proj = polyspace.Project;
  proj.Configuration = polyspace.ModelLinkOptions(modelName);
  ```

  Use the options to analyze the code generated from the model.

### Results — Analysis results

`polyspace.BugFinderResults` or `polyspace.CodeProverResults` object

Results of Polyspace analysis. When you create a `polyspace.Project` object, this property is initially empty. The property is populated only after you execute the `run` method of the object. Depending on the argument to the `run` method, `'bugFinder'` or `'codeProver'`, the property is implemented as a `polyspace.BugFinderResults` object or `polyspace.CodeProverResults` object.

To read the results, use these methods of the `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object:

- `getSummary`: Obtain a summarized format of the results into a MATLAB table.

  ```
  proj = polyspace.Project;
  proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
      'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
  proj.Configuration.ResultsDir = fullfile(pwd,'results');

  run(proj, 'bugFinder');

  resObj = proj.Results;
  resTable = getSummary(resObj, 'defects');
  ```

For more information, see getSummary.

- getResults: Obtain the full results or a more readable format into a MATLAB table.

  ```
  proj = polyspace.Project;
  proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
      'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
  proj.Configuration.ResultsDir = fullfile(pwd,'results');

  run(proj, 'bugFinder');

  resObj = proj.Results;
  resTable = getResults(resObj, 'readable');
  ```

  For more information, see getResults.

## Object Functions

run Run a Polyspace analysis

## Examples

### Check for Bugs

Run a Polyspace Bug Finder analysis on the example file numerical.c. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a results subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
resObj = proj.Results;
bfSummary = getSummary(resObj, 'defects');
```

### Prove Absence of Run-Time Errors

Run a Polyspace Code Prover analysis on the example file single_file_analysis.c. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a results subfolder of the current working folder.
- Specify that a main function must be generated, if the function does not exist in the source code.

```
proj = polyspace.Project

% Configure analysis
```

```
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;


% Run analysis
cpStatus = run(proj, 'codeProver');

% Read results
resObj = proj.Results;
cpSummary = getSummary(resObj, 'runtime');
```

### Check for Bugs and MISRA C:2012 Violations

Run a Polyspace Bug Finder analysis on the example file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Enable checking of MISRA C:2012 rules. Check for the mandatory rules only.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory';

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
resObj = proj.Results;
defectsSummary = getSummary(resObj, 'defects');
misraSummary = getSummary(resObj, 'misraC2012');
```

## Version History

**Introduced in R2017b**

## See Also

**Topics**
“Run Polyspace Analysis by Using MATLAB Scripts”
“Generate MATLAB Scripts from Polyspace User Interface”
“Troubleshoot Polyspace Analysis from MATLAB”
“Integrate Polyspace Server Products with MATLAB”

# polyspace.Options class

**Package:** polyspace

Create object for running Polyspace analysis on handwritten code

## Description

Create an object that specifies Polyspace options. When running a Polyspace analysis from MATLAB, specify the configuration by using this options object. To specify source files and customize analysis options, change the object properties.

polyspace.Options object apply to handwritten code. To analyze model-generated code (using the Polyspace desktop products), use polyspace.ModelLinkOptions instead.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” or “Integrate Polyspace Server Products with MATLAB”.

## Construction

opts = polyspace.Options creates an object whose properties correspond to options for running a Polyspace analysis.

proj = polyspace.Project creates a polyspace.Project object. The object has a property Configuration, which is a polyspace.Options object.

opts = polyspace.Options(lang) creates a Polyspace options object with options that are applicable to the language lang.

opts = polyspace.loadProject(projectFile) creates a Polyspace options object from an existing Polyspace project projectFile. You set the options in your project in the Polyspace user interface and create the options object from that project for programmatically running the analysis.

### Input Arguments

**lang — Language of analysis**
'C-CPP' (default) | 'C' | 'CPP'

The language of the analysis specified as 'C-CPP', 'C', or 'CPP'. This argument determines the object properties.

Data Types: char

**projectFile — Name of .psprj file**
character vector

Name of Polyspace project file with extension .psprj, specified as a character vector.

If the file is not in the current folder, projectFile must include a full or relative path. To identify the current folder, use pwd. To change the current folder, use cd.

**Note** You cannot use the `loadProject` method on a project file that is created from a build command by using `polyspace-configure`.

Example: `'C:\projects\myProject.psprj'`

## Properties

The object properties correspond to the analysis options for Polyspace projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS/UNIX command-line name. For syntax details, see polyspace.Project.Configuration properties.

## Methods

| | |
|---|---|
| copyTo | Copy common settings between Polyspace options objects |
| generateProject | Generate psprj project from options object |
| toScript | Add Polyspace options object definition to a script |

## Examples

### Customize and Run Analysis

Create a Polyspace analysis options object and customize the properties. Then, run an analysis.

Create object and customize properties. In case you do not have write access to your current folder, a temporary folder is being used for storing analysis results.

```
sources = fullfile(polyspaceroot, 'polyspace','examples','cxx','Bug_Finder_Example',...
    'sources','numerical.c');
opts = polyspace.Options();
opts.Prog = 'MyProject';
opts.Sources = {sources};
opts.TargetCompiler.Compiler = 'gnu4.7';
opts.ResultsDir = tempname;
```

Run a Bug Finder analysis. To run a Code Prover analysis, use `polyspaceCodeProver` instead of `polyspaceBugFinder`.

```
results = polyspaceBugFinder(opts);
```

With the Polyspace Server products, you can use the functions `polyspaceBugFinderServer` or `polyspaceCodeProverServer`.

Open the results in the Polyspace user interface of the desktop products.

```
polyspaceBugFinder('-results-dir',opts.ResultsDir);
```

### Run Polyspace by Generating a Project File

Create a Polyspace analysis options object and customize the properties. Then, run a Bug Finder analysis.

Create object and customize properties.

```
sources=fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example',...
    'sources','numerical.c');
opts = polyspace.Options();
opts.Prog = 'MyProject';
opts.Sources = {sources};
opts.TargetCompiler.Compiler = 'gnu4.7';
opts.ResultsDir = tempname;
```

Generate a Polyspace project, name it using the `Prog` property, and open the project in the Polyspace interface.

```
psprj = opts.generateProject(opts.Prog);
polyspaceBugFinder(psprj);
```

You can also analyze the project from the command line. Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceBugFinder(psprj, '-nodesktop');
polyspaceBugFinder('-results-dir',opts.ResultsDir);
```

## Alternatives

If you are analyzing code generated from a model, use `polyspace.ModelLinkOptions` instead.

# Version History

**Introduced in R2017a**

## See Also

`polyspace.ModelLinkOptions` | `polyspace.Project` | `polyspaceBugFinder` | `polyspaceBugFinderServer`

**Topics**

"Run Polyspace Analysis by Using MATLAB Scripts"
"Generate MATLAB Scripts from Polyspace User Interface"
"Integrate Polyspace Server Products with MATLAB"

# polyspace.ModelLinkOptions class

**Package:** `polyspace`

Create a project configuration object for running Polyspace analysis on generated code

## Description

Run a Polyspace analysis from MATLAB by using a project configuration object. To specify source files and customize analysis options, change the object properties.

This class is intended for model-generated code. If you are analyzing handwritten code, use `polyspace.Options` instead.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

## Construction

`psprjConfig = polyspace.ModelLinkOptions` creates a project configuration object that is configured for running a Polyspace analysis on generated code.

`psprjConfig = polyspace.ModelLinkOptions(lang)` creates a project configuration object that is configured to run analysis on code generated in the language `lang`.

`psprjConfig = polyspace.ModelLinkOptions(model)` creates a project configuration object that is configured by using model specific information from the Simulink model `model`. Prior to extracting options from `model`, you must load the model and generate code from it.

`psprjConfig = polyspace.ModelLinkOptions(model, psOpt)` creates a model-specific project configuration object that is configured by using the Polyspace analysis options specified in `psOpt`.

`psprjConfig = polyspace.ModelLinkOptions(model, psOpt, asModelRef)` creates a project configuration object that uses `asModelRef` to specify which type of generated code to analyze—standalone code or model reference code.

### Input Arguments

#### `lang` — Language of analysis

`'C-CPP'` (default) | `'C'` | `'CPP'`

The language of the analysis specified as `'C-CPP'`, `'C'`, or `'CPP'`. This argument determines the object properties.

#### `model` — Model or subsystem name

character vector

Name or path to model or subsystem, specified as a character vector.

Prior to extracting options from the model, you must:

1. Load the model. Use `load_system` or `open_system`.
2. Generate code from the model. Use `slbuild` or `slbuild`.

Example: `'psdemo_model_link_sl'`

**`psOpt` — Polyspace analysis options object**
`pslinkoptions` object

An object containing the options that you use for the Polyspace analysis. You create this by calling the function `pslinkoptions`. You can customize the options object by changing the properties of the `psOpt` object.

Example: `psOpt = pslinkoptions(model)` where `model` is the name of a Simulink model.

**`asModelRef` — Indicator for model reference analysis**
`false` (default) | `true`

Indicator for model reference analysis, specified as true or false.

- To analyze generated code used or called elsewhere, set the flag `asModelRef` to `true`. This option is equivalent to choosing **Analyze Code from** > **Code Generated as Model Reference** on the Polyspace tab in the Simulink toolstrip.
- To analyze code that is generated to be used as stand-alone code, set the flag `asModelRef` to `false`. This option is equivalent to choosing **Analyze Code from** > **Code Generated as Top model** on the Polyspace tab in the Simulink toolstrip.

Data Types: `logical`

## Properties

The object properties correspond to the configuration options for Polyspace projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS command-line name. For syntax details, see polyspace.ModelLinkOptions.

## Methods

| | |
|---|---|
| copyTo | Copy common settings between Polyspace options objects |
| generateProject | Generate psprj project from options object |
| toScript | Add Polyspace options object definition to a script |

## Examples

### Script Analysis of Model Generated Code

This example shows how to customize and run an analysis on code generated from a model.

Generate code from the model `rtwdemo_roll`. Before code generation, set a system target file appropriate for code analysis. See also “Recommended Model Configuration Parameters for Polyspace Analysis”.

```
modelName = 'rtwdemo_roll';
[TEMPDIR, CGDIR] = rtwdemodir();
```

```
load_system(modelName);

% Set parameters for Embedded Coder target
set_param(modelName, 'SystemTargetFile', 'ert.tlc');
set_param(modelName,'Solver','FixedStepDiscrete');
set_param(modelName,'SupportContinuousTime','on');
set_param(modelName,'LaunchReport','off');
set_param(modelName,'InitFltsAndDblsToZero','on');

slbuild(modelName);
```

Associate a `polyspace.ModelLinkOptions` object with the model. A subset of the object properties are set from the configuration parameters associated with the model. The other properties take their default values. For details on the configuration parameters, see “Bug Finder Analysis in Simulink”.

```
psprjCfg = polyspace.ModelLinkOptions(modelName);
```

Change the property values if needed. For instance, you can specify that the analysis must check for all MISRA C: 2012 violations and generate a PDF report of the results. You can also specify a folder for the analysis results.

```
psprjCfg.CodingRulesCodeMetrics.EnableMisraC3 = true;
psprjCfg.CodingRulesCodeMetrics.MisraC3Subset = 'all';
psprjCfg.MergedReporting.EnableReportGeneration = true;
psprjCfg.MergedReporting.ReportOutputFormat = 'PDF';
psprjCfg.ResultsDir = 'newResfolder';
```

Create a `polyspace.Project` object. Associate the `Configuration` property of this object to the options that you previously specified.

```
proj = polyspace.Project;
proj.Configuration = psprjCfg;
```

Optionally, you might want to submit the analysis to a remote server to run batches of analysis together. To setup analysis in batch mode, set the property `BatchBugFinder` or `BatchCodeProver` to `true`, depending on your analysis. Then, specify the head node that manages the Polyspace analysis jobs:

```
% Setup batch analysis
psprjCfg.MergedComputingSettings.BatchBugFinder = true;
psprjCfg.Advanced.Additional = '-scheduler NodeID';
```

Run analysis and open results.

```
cpStatus = proj.run('codeProver');
proj.Results.getResults('readable');
```

### Analyze Code Generated as Model Reference

This example shows how to analyze generated code used as a callable entity in another model or code.

Load the Simulink model `rtwdemo_roll` and configure it for a Polyspace analysis. For details, see “Recommended Model Configuration Parameters for Polyspace Analysis” for details.

```
% Make directory for code generation
[TEMPDIR, CGDIR] = rtwdemodir();
```

```
%  Specify model name
model = 'rtwdemo_roll';
% Load the model
load_system(model);
% Configure the model for generating code
set_param(model, 'SystemTargetFile', 'ert.tlc');
set_param(model,'MatFileLogging','off');
set_param(model,'GenerateComments','on');
set_param(model,'Solver','FixedStepDiscrete');
set_param(model,'LaunchReport','off');
```

To generate code as a model reference from the Simulink model, use `slbuild`. Set the `buildspec` parameter to `'ModelReferenceCoderTarget'`.

```
slbuild(model,'ModelReferenceCoderTarget');
```

To configure the Polyspace analysis of the generated code, create an options object `psOpt` by using the function `pslinkoptions`. Change the properties of the object as needed. For instance, to run a **Code Prover** analysis, set the `Verificationmode` to `'CodeProver'`.

```
psOpt = pslinkoptions(model);
psOpt.VerificationMode = 'CodeProver';
```

To run a Polyspace analysis, create and configure a Polyspace project configuration object.

- To create the Polyspace project configuration object, use the function `polyspace.ModelLinkOptions`.
- To associate the Polyspace analysis options with the project configuration, set the object `psOpt` as the second argument in `polyspace.ModelLinkOptions()`.
- To specify that the generated code must be analyzed as a model reference, specify the third argument as `'true'`.

For instance:

```
psprjCfg = polyspace.ModelLinkOptions(model, psOpt,true);
```

To configure the Polyspace project, change the properties of the `psprjCfg` object. For instance, to enable checkers for the mandatory MISRA C: 2012 rules and to generate a PDF report of the results, use:

```
psprjCfg.CodingRulesCodeMetrics.EnableMisraC3 = true;
psprjCfg.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory-required';
psprjCfg.MergedReporting.EnableReportGeneration = true;
psprjCfg.MergedReporting.ReportOutputFormat = 'PDF';
```

For convenience, you can specify a separate result folder.

```
psprjCfg.ResultsDir = 'newResfolder';
```

Create a Polyspace project by using `polyspace.Project` and associate the project configuration with it.

```
proj = polyspace.Project;
proj.Configuration = psprjCfg;
```

Run the Polyspace analysis by using the `run` function of the object `proj`.

```
cpStatus = proj.run('codeProver');
```

Because you enabled PDF report generation, the result of the Polyspace analysis is reported in a PDF file, which can be found in *newResfolder/Polyspace-Doc*. To view the results in a MATLAB table, use:

```
result = proj.Results.getResults('readable');
```

## Alternatives

If you are analyzing handwritten code, use a `polyspace.Project` object directly. Alternatively, use a `polyspace.Options` object.

# Version History

**Introduced in R2017a**

## See Also

`polyspace.Options` | `polyspace.Project` | `polyspaceBugFinder` | `pslinkrun`

**Topics**

"Run Polyspace Analysis by Using MATLAB Scripts"
polyspace.ModelLinkOptions Properties

# polyspace.BugFinderOptions class

**Package:** `polyspace`

(Removed) Create Polyspace Bug Finder object for handwritten code

**Note** This class is removed. Use `polyspace.Options` instead. See “Version History”.

## Description

Customize a Polyspace Bug Finder analysis from MATLAB by creating a Bug Finder options object. To specify source files and customize analysis options, change the object properties.

If you are analyzing model-generated code, use `polyspace.ModelLinkBugFinderOptions` instead.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

## Construction

`opts = polyspace.BugFinderOptions` creates a Bug Finder options object with available options.

`opts = polyspace.BugFinderOptions(lang)` creates a Bug Finder options object with options that are applicable for the language `lang`.

### Input Arguments

**`lang` — Language of analysis**
`'C-CPP'` (default) | `'C'` | `'CPP'`

The language of the analysis specified as `'C-CPP'`, `'C'`, or `'CPP'`. This argument determines which properties the object has.

## Properties

The object properties are the analysis options for Polyspace Bug Finder projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS/UNIX command-line name. For syntax details, see polyspace.Options.

## Methods

| | |
|---|---|
| copyTo | Copy common settings between Polyspace options objects |
| generateProject | Generate psprj project from options object |
| toScript | Add Polyspace options object definition to a script |

## Examples

### Customize and Run Analysis

Create a Bug Finder analysis options object and customize the properties. Then, run an analysis.

Create object and customize properties.

```
sources = fullfile(polyspaceroot, 'polyspace','examples','cxx','Bug_Finder_Example',...
    'sources','numerical.c');
optsBF = polyspace.BugFinderOptions();
optsBF.Prog = 'MyProject';
optsBF.Sources = {sources};
optsBF.TargetCompiler.Compiler = 'gnu4.7';
optsBF.ResultsDir = tempname;
```

Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceBugFinder(optsBF);
polyspaceBugFinder('-results-dir',optsBF.ResultsDir);
```

### Run Polyspace by Generating a Project File

Create a Bug Finder analysis options object and customize the properties. Then, run an analysis.

Create object and customize properties.

```
sources = fullfile(polyspaceroot, 'polyspace','examples','cxx','Bug_Finder_Example',...
    'sources','numerical.c');
optsBF = polyspace.BugFinderOptions();
optsBF.Prog = 'MyProject';
optsBF.Sources = {sources};
optsBF.TargetCompiler.Compiler = 'gnu4.7';
optsBF.ResultsDir = tempname;
```

Generate a Polyspace project, name it using the `Prog` property, and open the project in the Polyspace interface.

```
psprj = generateProject(optsBF, optsBF.Prog);
polyspaceBugFinder(psprj);
```

Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceBugFinder(psprj, '-nodesktop');
polyspaceBugFinder('-results-dir',optsBF.ResultsDir);
```

## Alternatives

If you are analyzing generated code, use `polyspace.ModelLinkOptions` instead.

# Version History

**Introduced in R2016b**

### R2022b: Class removed

*Errors starting in R2022b*

Using this class to set Polyspace analysis options results in an error. Use the class `polyspace.Options` instead. For instance, replace this code:

```
optsBF = polyspace.BugFinderOptions();
```

with this:

```
optsBF = polyspace.Options();
```

### R2017a: Class to be removed

*Warns starting in R2017a*

If you use this class to set Polyspace analysis options, MATLAB produces an warning.

## See Also

polyspace.Options | `polyspace.ModelLinkBugFinderOptions` | `polyspaceBugFinder`

### Topics

"Run Polyspace Analysis by Using MATLAB Scripts"

# polyspace.DefectsOptions class

**Package:** `polyspace`

Create custom list of defects to check

## Description

Create a custom list of defects to check in a Polyspace analysis.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” or “Integrate Polyspace Server Products with MATLAB”.

## Construction

`defectsList = polyspace.DefectsOptions` creates the defect options object `defectsList`. You can customize the list of active defects by changing the properties.

## Properties

An object is created with supported defects as properties. The defects are listed by their command-line name. See “Short Names of Bug Finder Defect Groups and Defect Checkers”.

By default, all defects are turned off. To turn on a defect, set the defect to true. For example:

```
defectsList = polyspace.DefectsOptions;
defectsList.FLOAT_ZERO_DIV = true;
```

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

## Examples

### Customize List of Defects to Check

Customize the list of defects checked during a Polyspace Bug Finder analysis.

Create two objects: a `polyspace.DefectsOptions` object for setting coding rules and a `polyspace.Project` object for running the Polyspace analysis.

```
defectsList = polyspace.DefectsOptions;
proj = polyspace.Project;
```

Enable the numerical defects.

```
defectsList.FLOAT_ZERO_DIV = true;
defectsList.INT_ZERO_DIV = true;
defectsList.FLOAT_ABSORPTION = true;
defectsList.BITWISE_NEG = true;
defectsList.FLOAT_CONV_OVFL = true;
defectsList.FLOAT_OVFL = true;
defectsList.INT_CONV_OVFL = true;
defectsList.INT_OVFL = true;
defectsList.FLOAT_STD_LIB = true;
defectsList.INT_STD_LIB = true;
defectsList.SHIFT_NEG = true;
defectsList.SHIFT_OVFL = true;
defectsList.SIGN_CHANGE = true;
defectsList.UINT_CONV_OVFL = true;
defectsList.UINT_OVFL = true;
defectsList.BAD_PLAIN_CHAR_USE = true;
```

Add the customized list of defects to the `Configuration` property of the `polyspace.Project` object.

```
proj.Configuration.BugFinderAnalysis.CheckersList = defectsList;
proj.Configuration.BugFinderAnalysis.CheckersPreset = 'custom';
```

You can now use the `polyspace.Project` object to run the analysis.

## Version History

**Introduced in R2016b**

## See Also

`polyspace.Project` | `polyspace.Options` | `polyspace.ModelLinkOptions` | `polyspace.CodingRulesOptions`

### Topics

"Short Names of Bug Finder Defect Groups and Defect Checkers"

# polyspace.ModelLinkBugFinderOptions class

**Package:** `polyspace`

(Removed) Create Polyspace Bug Finder object for generated code

**Note** This class is removed. Use `polyspace.ModelLinkOptions` instead. Use `polyspace.ModelLinkOptions` instead. See “Version History”.

## Description

Customize a Polyspace Bug Finder analysis from MATLAB by creating a Bug Finder options object. To specify source files and customize analysis options, change the object properties.

This class is intended for model-generated code. If you are analyzing handwritten code, use `polyspace.BugFinderOptions` instead.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

## Construction

`opts = polyspace.BugFinderOptions` creates a Bug Finder options object for generated code with available options for C/C++ generated code.

## Properties

The object properties are the analysis options for Polyspace Bug Finder model link projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS command-line name. For syntax details, see polyspace.ModelLinkOptions.

## Methods

| | |
|---|---|
| copyTo | Copy common settings between Polyspace options objects |
| generateProject | Generate psprj project from options object |
| toScript | Add Polyspace options object definition to a script |

## Examples

### Script Analysis of Model Generated Code

This example shows how to customize and run an analysis on model generated code with MATLAB functions and objects.

Create a custom configuration that checks MISRA C 2012 rules and generates a PDF report.

```
opts = polyspace.ModelLinkBugFinderOptions();
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';
opts.MergedReporting.ReportOutputFormat = 'PDF';
opts.MergedReporting.EnableReportGeneration = true;
```

Generate code from `psdemo_model_link_sl`.

```
[TEMPDIR, CGDIR] = rtwdemodir();
model = 'psdemo_model_link_sl';
load_system(model);
slbuild(model);
```

Add the configuration to `pslinkoptions` object.

```
prjfile = opts.generateProject('model_link_opts');
mlopts = pslinkoptions(model);
mlopts.EnablePrjConfigFile = true;
mlopts.PrjConfigFile = prjfile;
mlopts.VerificationMode = 'BugFinder';
```

Run analysis.

```
[polyspaceFolder, resultsFolder] = pslinkrun(model);
```

## Alternatives

If you are analyzing handwritten code, use `polyspace.Options` instead.

# Version History

**R2022b: Class Removed**
*Errors starting in R2022b*

Using this class to set Polyspace analysis options results in an error. Use the class `polyspace.ModelLinkOptions` instead. For instance, replace this code:

```
opts = polyspace.ModelLinkBugFinderOptions();
```

with this:

```
opts = polyspace.ModelLinkOptions();
```

**R2017a: Class to be removed**
*Warns starting in R2017a*

If you use this class to set Polyspace analysis options, MATLAB produces an warning.

## See Also

polyspace.ModelLinkOptions | `polyspace.BugFinderOptions` | `polyspaceBugFinder` | `pslinkrun`

**Topics**

"Run Polyspace Analysis by Using MATLAB Scripts"

# polyspace.GenericTargetOptions class

**Package:** `polyspace`

Create a generic target configuration

## Description

Create a custom target for a Polyspace analysis if your target processor does not match one of the predefined targets,.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” or “Integrate Polyspace Server Products with MATLAB”.

## Construction

`genericTarget = polyspace.GenericTargetOptions` creates a generic target that you can customize. To specify the sizes and alignment of data types, change the properties of the object. For instance:

```
target = polyspace.GenericTargetOptions;
target.CharNumBits = 16;
```

## Properties

For more details about any of the properties below, see `Generic target options`.

### `Alignment` — Largest alignment of struct or array objects

32 (default) | 16 | 8

Largest alignment of struct or array objects, specified as `32`, `16`, or `8`. Comparable with the DOS/UNIX command-line option `-align`.

Example: `target.Alignment = 8`

### `CharNumBits` — Define the number of bits for a `char`

8 (default) | 16

Define the number of bits for a `char`, specified as `8` or `16`. Comparable with the DOS/UNIX command-line option `-char-is-16bits`.

Example: `target.CharNumBits = 16`

### `DoubleNumBits` — Define the number of bits for a `double`

32 (default) | 64

Define the number of bits for a `double`, specified as `32` or `64`. Comparable with the DOS/UNIX command-line option `-double-is-64bits`.

Example: `target.DoubleNumBits = 64`

### `Endianness` — Endianness of target architecture

`little` (default) | `big`

Endianness of target architecture, specified as `little` or `big`. Comparable with the DOS/UNIX command-line options `-little-endian` or `-big-endian`.

Example: `target.Endianess = 'big'`

### `IntNumBits` — Define the number of bits for an `int`

16 (default) | 32

Define the number of bits for an `int`, specified as `16` or `32`. Comparable with the DOS/UNIX command-line option `-int-is-32bits`.

Example: `target.IntNumBits = 32`

### `LongLongNumBits` — Define the number of bits for a `long long`

32 (default) | 64

Define the number of bits for a `long long`, specified as `32` or `64`. Comparable with the DOS/UNIX command-line option `-long-long-is-64bits`.

Example: `target.LongNumBits = 64`

### `LongNumBits` — Define the number of bits for a `long`

32 (default)

Define the number of bits for a `long`, specified as `32`. Comparable with the DOS/UNIX command-line option `-long-is-32bits`.

Example: `target.LongNumBits = 32`

### `PointerNumBits` — Define the number of bits for a pointer

16 (default) | 24 | 32

Define the number of bits for a pointer, specified as `16`, `24`, or `32`. Comparable with the DOS/UNIX command-line options `-pointer-is-24bits` and `-pointer-is-32bits`.

Example: `target.PointerNumBits = 32`

### `ShortNumBits` — Define the number of bits for a `short`

16 (default) | 8

Define the number of bits for an `int`, specified as `16` or `8`. Comparable with the DOS/UNIX command-line option `-short-is-8bits`.

Example: `target.ShortNumBits = 8`

### `SignOfChar` — Default sign of plain `char`

`signed` (default) | `unsigned`

Default sign of plain `char`, specified as `signed` or `unsigned`. Comparable with the DOS/UNIX command-line option `-default-sign-of-char`.

Example: `target.SignOfChar = 'unsigned'`

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

## Examples

### Customize Generic Target Settings

Use a custom target for the Polyspace analysis.

Create two objects: a `polyspace.GenericTargetOptions` object for creating a custom target and a `polyspace.Project` object for running the Polyspace analysis.

```
target = polyspace.GenericTargetOptions;
proj = polyspace.Project;
```

Customize the generic target.

```
target.Endianess = 'big';
target.LongLongNumBits = 64;
target.ShortNumBits = 8;
```

Add the custom target to the `Configuration` property of the `polyspace.Project` object.

```
proj.Configuration.TargetCompiler.Target = target;
```

You can now use the `polyspace.Project` object to run the analysis.

`polyspace.Project` | `polyspace.Options` | `polyspace.ModelLinkOptions` | `polyspace.CodingRulesOptions` | `Generic target options`

# Version History

**Introduced in R2016b**

# polyspace.CodingRulesOptions class

**Package:** polyspace

Create custom list of coding rules to check

## Description

Create a custom list of coding rules to check in a Polyspace analysis.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” or “Integrate Polyspace Server Products with MATLAB”.

## Construction

ruleList = polyspace.CodingRulesOptions(RuleSet) creates the coding rules object ruleList for the RuleSet coding rule set. Set the active rules in the coding rules object.

### Input Arguments

**RuleSet — Standard coding rule set**
misraC (default) | misraC2012 | misraAcAgc | misraCpp | jsf | certC | certCpp | iso17961 | autosarCpp14 | guidelines | cwe

Standard coding rule set specified as one of the coding rule acronyms.

Example: 'misraCpp'

Data Types: char

## Properties

For each coding rule set, an object is created with all supported rules divided into sections. By default, all rules are on. To turn off a rule, set the rule to false. For example:

```
misraRules = polyspace.CodingRulesOptions('misraC');
misraRules.Section_20_Standard_libraries.rule_20_1 = false;
```

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

## Examples

### Customize List of Coding Rules to Check

Customize the coding rules that are checked in a Polyspace analysis. Since all rules are enabled by default, you can create a custom subset by disabling some rules.

Create two objects: a `polyspace.CodingRulesOptions` object for setting coding rules and a `polyspace.Project` object for running the Polyspace analysis.

```
misraRules = polyspace.CodingRulesOptions('misraC2012');
proj = polyspace.Project;
```

Customize the coding rule list by turning off rules 2.1-2.7.

```
misraRules.Section_2_Unused_code.rule_2_1 = false;
misraRules.Section_2_Unused_code.rule_2_2 = false;
misraRules.Section_2_Unused_code.rule_2_3 = false;
misraRules.Section_2_Unused_code.rule_2_4 = false;
misraRules.Section_2_Unused_code.rule_2_5 = false;
misraRules.Section_2_Unused_code.rule_2_6 = false;
misraRules.Section_2_Unused_code.rule_2_7 = false;
```

Add the customized list of coding rules to the `Configuration` property of the `polyspace.Project` object.

```
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = misraRules;
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;
```

You have to enable checkers selection by file because the Polyspace run uses an XML file underneath to enable the coding rule checkers. The XML file is saved in a `.settings` subfolder of the results folder.

You can now use the `polyspace.Project` object to run the analysis. For instance, you can enter:

```
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
run(proj, 'bugfinder');
```

### Create Coding Rules Object Using Rule Numbers to Enable

Suppose that you want to specify a subset of MISRA C: 2012 rules for the analysis. Instead of enumerating rules that you want disabled, you can specify the rules that you want to keep enabled. You can also specify the rule numbers only without the MISRA C: 2012 sections containing the rules.

Specify the rule numbers in a cell array to the `createRulesObject` function defined as follows.

```
function rulesObject = createRulesObject(rulesToEnable)

%% This function takes a cell array of MISRA C:2012 rules and returns
%% a polyspace.CodingRulesOptions object with the rules enabled.
%% Example input argument: {'2.7', '3.1'}

    rulesObject = polyspace.CodingRulesOptions('misraC2012');

    % Coding Standards documents have many sections. Loop over all
    % sections.
    ruleSections = properties(rulesObject);
    for i=1:length(ruleSections)
         sectionName = ruleSections{i};
         rulesInSection = properties(rulesObject.(sectionName));
```

```
        % Loop over all rules in a section, enable or disable rule based
        % on input
        for j=1:length(rulesInSection)
            ruleNumberAsProperty = rulesInSection{j};
            ruleNumber = strrep(strrep(ruleNumberAsProperty,'rule_',''),'_','.');
            if(any(strcmp(rulesToEnable,ruleNumber)))
                rulesObject.(sectionName).(ruleNumberAsProperty)=1;
            else
                rulesObject.(sectionName).(ruleNumberAsProperty)=0;
            end
        end
    end
end
```

For instance, to enable rules 1.1 and 2.2, enter:

```
createRulesObject({'1.1','2.2'})
```

## Version History

**Introduced in R2016b**

### See Also

polyspace.Project | polyspace.Options | polyspace.ModelLinkOptions

# polyspace.BugFinderResults

Read Polyspace Bug Finder results from MATLAB

## Description

Read Polyspace Bug Finder analysis results to MATLAB tables by using this object. You can obtain a high-level overview of results or details such as each instance of a defect.

**Note** Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink” or “Integrate Polyspace Server Products with MATLAB”.

## Creation

### Syntax

```
resObj = polyspace.BugFinderResults(resultsFolder)
proj = polyspace.Project; resObj = proj.Results;
```

#### Description

`resObj = polyspace.BugFinderResults(resultsFolder)` creates an object for reading a specific set of Bug Finder results into MATLAB tables. Use the object methods to read the results.

`proj = polyspace.Project; resObj = proj.Results;` creates a `polyspace.Project` object with a `Results` property. If you run a Bug Finder analysis, this property is a `polyspace.BugFinderResults` object.

#### Input Arguments

**`resultsFolder` — Name of result folder**
character vector

Name of result folder, specified as a character vector. The folder must directly contain the results file with extension `.psbf`. Even if the results file resides in a *subfolder* of the specified folder, it cannot be accessed.

If the folder is not in the current folder, `resultsFolder` must include a full or relative path.

Example: `'C:\Polyspace\Results\'`

### Object Functions

| | |
|---|---|
| getSummary | View number of Polyspace results organized by results type (Bug Finder) or color and file (Code Prover) |
| getResults | View all instances of Bug Finder or Code Prover results |

## Examples

### Read Existing Results to MATLAB Tables

This example shows how to read Bug Finder analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath=fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example',...
'Module_1','BF_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.BugFinderResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary (resObj);
resTable = getResults (resObj);
```

### Run Analysis and Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project;

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace',...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
resObj = proj.Results;
bfSummary = getResults(resObj, 'readable');
```

# Version History

**Introduced in R2017a**

# pslinkoptions Properties

Properties for the `pslinkoptions` object

## Description

You can create a `pslinkoptions` object to customize your analysis at the command-line. Use these properties to specify configuration options, where and how to store results, additional files to include, and data range modes.

### Properties

**Configuration Options**

**`VerificationSettings` — Coding rule and configuration settings for C code**
`'PrjConfig'` (default) | `'PrjConfigAndMisraAGC'` | `'PrjConfigAndMisra'` | `'PrjConfigAndMisraC2012'` | `'MisraAGC'` | `'Misra'` | `'MisraC2012'`

Coding rule and configuration settings for C code specified as:

- `'PrjConfig'` – Inherit options from the project configuration.
- `'PrjConfigAndMisraAGC'` – Inherit options from the project configuration and enable MISRA AC AGC rule checking.
- `'PrjConfigAndMisra'` – Inherit options from the project configuration and enable MISRA C:2004 rule checking.
- `'PrjConfigAndMisraC2012'` – Inherit options from the project configuration and enable MISRA C:2012 guideline checking.
- `'MisraAGC'` – Enable MISRA AC AGC rule checking. This option runs only compilation and rule checking.
- `'Misra'` – Enable MISRA C:2004 rule checking. This option runs only compilation and rule checking.
- `'MisraC2012'` – Enable MISRA C:2012 rule checking. This option runs only compilation and guideline checking.

Example: `opt.VerificationSettings = 'PrjConfigAndMisraC2012'`

**`VerificationMode` — Polyspace mode**
`'BugFinder'` (default) | `'CodeProver'`

Polyspace mode specified as `'BugFinder'`, for a Bug Finder analysis, or `'CodeProver'`, for a Code Prover verification.

Example: `opt.VerificationMode = 'BugFinder';`

**`EnablePrjConfigFile` — Allow a custom configuration file**
`false` (default) | `true`

Allows a custom configuration file instead of the default configuration specified as true or false. Use the `PrjConfigFile` option to specify the configuration file.

Example: `opt.EnablePrjConfigFile = true;`

### `PrjConfigFile` — Custom configuration file

`''` (default) | full path to a `.psprj` file

Custom configuration file to use instead of the default configuration specified by the full path to a `.psprj` file. Use the `EnablePrjConfigFile` option to use this configuration file during your analysis.

Example: `opt.PrjConfigFile = 'C:\Polyspace\config.psprj';`

### `CheckConfigBeforeAnalysis` — Configuration check before analysis

`'OnWarn'` (default) | `'OnHalt'` | `'Off'`

This property sets the level of configuration checking done before the analysis starts. The configuration check before analysis is specified as:

- **`'Off'`** — Checks only for errors. Stops if errors are found.
- **`'OnWarn'`** — Stops for errors. Displays a message for warnings.
- **`'OnHalt'`** — Stops for errors and warnings.

Example: `opt.CheckConfigBeforeAnalysis = 'OnHalt';`

#### Results

### `ResultDir` — Results folder name and location

`'C:\Polyspace_Results\results_$ModelName$'` (default) | folder name | folder path

Results folder name and location specified as the local folder name or the folder path. This folder is where Polyspace writes the analysis results. This folder name can be either an absolute path or a path relative to the current folder. The text `$ModelName$` is replaced with the name of the original model.

Example: `opt.ResultDir = '\results_v1_$ModelName$';`

### `AddSuffixToResultDir` — Add unique number to the results folder name

`false` (default) | `true`

Add unique number to the results folder name specified as true or false. If true, a unique number is added to the end of every new result. Using this option helps you avoid overwriting the previous results folders.

Example: `opt.AddSuffixToResultDir = true;`

### `OpenProjectManager` — Open the Polyspace environment

`false` (default) | `true`

Open the Polyspace environment to monitor the progress of the analysis, specified as true or false. Afterward, you can review the results.

Example: `opt.OpenProjectManager = true;`

### `AddToSimulinkProject` — Add results to the open Simulink project

`false` (default) | `true`

Add your results to the currently open Simulink project, if any, specified as true or false. This option allows you to keep your Polyspace results organized with the rest of your project files. If a Simulink project is not open, the results are not added to a Simulink project.

Example: `opt.AddToSimulinkProject = true;`

#### Additional Files

### `EnableAdditionalFileList` — Allow an additional file list

`false` (default) | `true`

Allow an additional file list to be analyzed, specified as true or false. Use with the `AdditionalFileList` option.

Example: `opt.EnableAdditionalFileList = true;`

### `AdditionalFileList` — List of additional files to be analyzed

`{0x1 cell}` (default) | cell array of files

List of additional files to be analyzed specified as a cell array of files. Use with the `EnableAdditionalFileList` option to add these files to the analysis.

Example: `opt.AdditionalFileList = {'sources\file1.c', 'sources\file2.c'};`

Data Types: `cell`

#### Data Ranges

### `InputRangeMode` — Enable design range information

`'DesignMinMax'` (default) | `'FullRange'`

Enable design range information specified as `'DesignMinMax'`, to use data ranges defined in blocks and workspaces, or `'FullRange'`, to treat inputs as full-range values.

Example: `opt.InputRangeMode = 'FullRange';`

### `ParamRangeMode` — Enable constant parameter values

`'None'` (default) | `'DesignMinMax'`

Enable constant parameter values, specified as `'None'`, to use constant parameters values specified in the code, or `'DesignMinMax'` to use a range defined in blocks and workspaces.

Example: `opt.ParamRangeMode = 'DesignMinMax';`

### `OutputRangeMode` — Enable output assertions

`'None'` (default) | `'DesignMinMax'`

Enable output assertions specified by `'None'`, to not apply assertions, or `'DesignMinMax'` to apply assertions to outputs using a range defined in blocks and workspace.

Example: `opt.ParamRangeMode = 'DesignMinMax';`

#### Embedded Coder Only

### `ModelRefVerifDepth` — Depth of verification

`'Current model only'` (default) | `'1'` | `'2'` | `'3'` | `'All'`

Specify the depth for analyzing the models that are referenced by the current model.

- `'Current Model Only'`: Analyze only the top model without analyzing the referenced models. For instance, you might use this option when the referenced models are library models.
- `'1'`, `'2'`, or `'3'`: Analyze referenced models up to the specified depth in the reference hierarchy. For instance, to analyze the models that are referenced by the top model, specify the property

ModelRefVerifDepth as '1'. To analyze models that are referenced by the first level of references, specify this property as '2'.

- 'All': Analyze all referenced models with the current model.

*For Embedded Coder only*

Example: opt.ModelRefVerifDepth = '3';

### ModelRefByModelRefVerif — Model reference analysis mode

false (default) | true

Specify whether you want to analyze all referenced models together, or analyze the models individually.

- false: Analyze the top model and the referenced models together. For instance, you might want to use this option to check for integration or scaling issues.
- true: Analyze the top model and the referenced models individually.

*For Embedded Coder only*

Example: opt.ModelRefByModelRefVerif = true;

### CxxVerificationSettings — Coding rule and configuration settings for C++ code

'PrjConfig' (default) | 'PrjConfigAndMisraCxx' | 'PrjConfigAndJSF' | 'MisraCxx' | 'JSF'

Coding rule and configuration settings for C++ code specified as:

- 'PrjConfig' – Inherit options from project configuration and run complete analysis.
- 'PrjConfigAndMisraCxx' – Inherit options from project configuration, enable MISRA C++ rule checking, and run complete analysis.
- 'PrjConfigAndJSF' – Inherit options from project configuration, enable JSF rule checking, and run complete analysis.
- 'MisraCxx' – Enable MISRA C++ rule checking, and run compilation phase only.
- 'JSF' – Enable JSF rule checking, and run compilation phase only.

*Only for Embedded Coder*

Example: opt.CxxVerificationSettings = 'MisraCxx';

#### TargetLink Only

### AutoStubLUT — Lookup Table code usage

false (default) | true

Lookup Table code usage, specified as true or false.

- true — use Lookup Table code during the analysis.
- false — stub Lookup Table code.

*Only for TargetLink*

Example: opts.AutoStubLUT = true;

## See Also

`pslinkoptions` | `pslinkrun`

# polyspace.Project.Configuration Properties

Customize Polyspace analysis of handwritten code with options object properties

## Description

To customize your Polyspace analysis, use these `polyspace.Options` or `polyspace.Project.Configuration` properties. Each property corresponds to an analysis option on the **Configuration** pane in the Polyspace user interface.

The properties are grouped using the same categories as the **Configuration** pane. This page only shows what values each property can take. For details about:

- The different options, see the analysis option reference pages.
- How to create and use the object, see `polyspace.Options` or `polyspace.Project`.

  Avoid using the classes `polyspace.BugFinderOptions` and `polyspace.CodeProverOptions` to set these properties. These classes will be removed in a future release.

Each property description below also highlights if the option affects only one of Bug Finder or Code Prover.

**Note** Some options might not be available depending on the language setting of the object. You can set the source code language (`Language`) to `'C'`, `'CPP'` or `'C-CPP'` during object creation, but cannot change it later.

### Properties

**Advanced**

**`Additional` — Additional flags for analysis**
character vector

Additional flags for analysis specified as a character vector.

For more information, see `Other`.

Example: `opts.Advanced.Additional = '-extra-flags -option -extra-flags value'`

**`PostAnalysisCommand` — Command or script software should execute after analysis finishes**
character vector

Command or script software should execute after analysis finishes, specified as a character vector.

For more information, see `Command/script to apply after the end of the code verification (-post-analysis-command)`.

Example: `opts.Advanced.PostAnalysisCommand = '"C:\Program Files\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"'`

#### BugFinderAnalysis (Affects Bug Finder Only)

**`CheckersList` — List of custom checkers to activate**
`polyspace.DefectsOptions` object | cell array of defect acronyms

*This property affects Bug Finder analysis only.*

List of custom checkers to activate specified by using the name of a `polyspace.DefectsOptions` object or a cell array of defect acronyms. To use this custom list in your analysis, set `CheckersPreset` to `custom`.

For more information, see `polyspace.DefectsOptionspolyspace.DefectsOptions`.

Example: `defects = polyspace.DefectsOptions; opts.BugFinderAnalysis.CheckersList = defects`

Example: `opts.BugFinderAnalysis.CheckersList = {'INT_ZERO_DIV','FLOAT_ZERO_DIV'}`

**`CheckersPreset` — Subset of Bug Finder defects**
`'default'` (default) | `'all'` | `'custom'`

*This property affects Bug Finder analysis only.*

Preset checker list, specified as a character vector of one of the preset options: `'default'`, `'all'`,or `'custom'`. To use `'custom'`, specify a value for the property `BugFinderAnalysis.CheckersList`.

For more information, see `Find defects (-checkers)`.

Example: `opts.BugFinderAnalysis.CheckersPreset = 'all'`

**`ChecksUsingSystemInputValues` — Activate stricter checks for system inputs**
false (default) | true

*This property affects Bug Finder analysis only.*

Activate stricter checks that consider all possible value for:

- Global variables.
- Reads of volatile variables.
- Returns of stubbed functions.
- Inputs to functions specified with **SystemInputsFrom**.

The analysis considers all possible values for a subset of **Numerical** and **Static memory** defects.

This property is equivalent to the **Run stricter checks considering all values of system inputs** check box in the Polyspace interface.

For more information, see `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`.

Example: `opts.BugFinderAnalysis.ChecksUsingSystemInputValues = true`

**`EnableCheckers` — Activate defect checking**
true (default) | false

*This property affects Bug Finder analysis only.*

Activate defect checking, specified as true or false. Setting this property to false disables all defects. If you want to disable defect checking but still get results, turn on coding rules checking or code metric checking.

This property is equivalent to the **Find defects** check box in the Polyspace interface.

Example: `opts.BugFinderAnalysis.EnableCheckers = false`

### `SystemInputsFrom` — List of functions for which you run stricter checks

`'auto'` (default) | `'uncalled'` | `'all'` | `'custom'`

*This property affects Bug Finder analysis only.*

Functions for which you want to run stricter checks that consider all possible values of the function inputs. Specify the list of functions as `'auto'`, `'uncalled'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of function names.

To enable this option, set `BugFinderAnalysis.ChecksUsingSystemInputValues = true`.

For more information, see `Consider inputs to these functions (-system-inputs-from)`.

Example: `opts.BugFinderAnalysis.SystemInputsFrom = 'custom=foo,bar'`

#### ChecksAssumption (Affects Code Prover Only)

### `AllowNegativeOperandInShift` — Allow left shift operations on a negative number

false (default) | true

*This property affects Code Prover analysis only.*

Allow left shift operations on a negative number, specified as true or false.

For more information, see `Allow negative operand for left shifts (-allow-negative-operand-in-shift)`.

Example: `opts.ChecksAssumption.AllowNegativeOperandInShift = true`

### `AllowNonFiniteFloats` — Incorporate infinities and/or NaNs

false (default) | true

*This property affects Code Prover analysis only.*

Incorporate infinities and/or NaNs, specified as true or false.

For more information, see `Consider non finite floats (-allow-non-finite-floats)`.

Example: `opts.ChecksAssumption.AllowNonFiniteFloats = true`

### `AllowPtrArithOnStruct` — Allow arithmetic on pointer to a structure field so that it points to another field

false (default) | true

*This property affects Code Prover analysis only.*

Allow arithmetic on pointer to a structure field so that it points to another field, specified as true or false.

For more information, see `Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)`.

Example: `opts.ChecksAssumption.AllowPtrArithOnStruct = true`

### `CheckInfinite` — Detect floating-point operations that result in infinities

`'allow'` (default) | `'warn-first'` | `'forbid'`

*This property affects Code Prover analysis only.*

Detect floating-point operations that result in infinities.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `Infinities (-check-infinite)`.

Example: `opts.ChecksAssumption.CheckInfinite = 'forbid'`

### `CheckNan` — Detect floating-point operations that result in NaN-s

`'allow'` (default) | `'warn-first'` | `'forbid'`

*This property affects Code Prover analysis only.*

Detect floating-point operations that result in NaN-s.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `NaNs (-check-nan)`.

Example: `opts.ChecksAssumption.CheckNan = 'forbid'`

### `CheckSubnormal` — Detect operations that result in subnormal floating point values

`'allow'` (default) | `'warn-first'` | `'warn-all'` | `'forbid'`

*This property affects Code Prover analysis only.*

Detect operations that result in subnormal floating point values.

For more information, see `Subnormal detection mode (-check-subnormal)`.

Example: `opts.ChecksAssumption.CheckSubnormal = 'forbid'`

### `DetectPointerEscape` — Find cases where a function returns a pointer to one of its local variables

false (default) | true

*This property affects Code Prover analysis only.*

Find cases where a function returns a pointer to one of its local variables, specified as true or false.

For more information, see `Detect stack pointer dereference outside scope (-detect-pointer-escape)`.

Example: `opts.ChecksAssumption.DetectPointerEscape = true`

### `DisableInitializationChecks` — Disable checks for noninitialized variables and pointers

false (default) | true

*This property affects Code Prover analysis only.*

Disable checks for noninitialized variables and pointers, specified as true or false.

For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

Example: `opts.ChecksAssumption.DisableInitializationChecks = true`

**`PermissiveFunctionPointer` — Allow type mismatch between function pointers and the functions they point to**
false (default) | true

*This property affects Code Prover analysis only.*

Allow type mismatch between function pointers and the functions they point to, specified as true or false.

For more information, see `Permissive function pointer calls (-permissive-function-pointer)`.

Example: `opts.ChecksAssumption.PermissiveFunctionPointer = true`

**`SignedIntegerOverflows` — Behavior of signed integer overflows**
`'forbid'` (default) | `'allow'` | `'warn-with-wrap-around'`

*This property affects Code Prover analysis only.*

Enable the check for signed integer overflows and the assumptions to make following an overflow specified as `'forbid'`, `'allow'`, or `'warn-with-wrap-around'`.

For more information, see `Overflow mode for signed integer (-signed-integer-overflows)`.

Example: `opts.ChecksAssumption.SignedIntegerOverflows = 'warn-with-wrap-around'`

**`SizeInBytes` — Allow a pointer with insufficient memory buffer to point to a structure**
false (default) | true

*This property affects Code Prover analysis only.*

Allow a pointer with insufficient memory buffer to point to a structure, specified as true or false.

For more information, see `Allow incomplete or partial allocation of structures (-size-in-bytes)`.

Example: `opts.ChecksAssumption.SizeInBytes = true`

**`StackUsage` — Calculate the estimated stack usage of your code**
false (default) | true

*This property affects Code Prover analysis only.*

Calculate stack usage metrics such as maximum or minimum stack usage and estimates of local variable size.

For more information, see `Calculate stack usage (-stack-usage)`.

Example: `opts.ChecksAssumption.StackUsage = true`

### `UncalledFunctionCheck` — Detect functions that are not called directly or indirectly from main or another entry-point function

`'none'` (default) | `'never-called'` | `'called-from-unreachable'` | `'all'`

*This property affects Code Prover analysis only.*

Detect functions that are not called directly or indirectly from main or another entry-point function, specified as `none`, `never-called`, `called-from-unreachable`, or `all`.

For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

Example: `opts.ChecksAssumption.UncalledFunctionCheck = 'all'`

### `UnsignedIntegerOverflows` — Behavior of unsigned integer overflows

`'allow'` (default) | `'forbid'` | `'warn-with-wrap-around'`

*This property affects Code Prover analysis only.*

Enable the check for unsigned integer overflows and the assumptions to make following an overflow, specified as `'forbid'`, `'allow'`, or `'warn-with-wrap-around'`.

For more information, see `Overflow mode for unsigned integer (-unsigned-integer-overflows)`.

Example: `opts.ChecksAssumption.UnsignedIntegerOverflows = 'allow'`

#### CodeProverVerification (Affects Code Prover only)

### `ClassAnalyzer` — Classes that you want to verify

`'all'` (default) | `'none'` | `'custom=`*`class1[,class2,...]`*`'`

*This property affects Code Prover analysis only.*

Classes that you want to verify, specified as `'all'`, `'none'`, or as a character array beginning with `custom=` followed by a comma-separated list of class names.

For more information, see `Class (-class-analyzer)`.

Example: `opts.CodeProverVerification.ClassAnalyzer = 'custom=myClass1,myClass2'`

### `ClassAnalyzerCalls` — Class methods that you want to verify

`'unused'` (default) | `'all'` | `'all-public'` | `'inherited-all'` | `'inherited-all-public'` | `'unused-public'` | `'inherited-unused'` | `'inherited-unused-public'` | `'custom=`*`method1[,method2,...]`*`'`

*This property affects Code Prover analysis only.*

Class methods that you want to verify, specified as one of the predefined sets or as a character array beginning with `custom=` followed by a comma-separated list of method names.

For more information, see `Functions to call within the specified classes (-class-analyzer-calls)`.

Example: `opts.CodeProverVerification.ClassAnalyzerCalls = 'unused-public'`

### `ClassOnly` — Analyze only class methods

false (default) | true

*This property affects Code Prover analysis only.*

Analyze only class methods, specified as true or false.

For more information, see `Analyze class contents only (-class-only)`.

Example: `opts.CodeProverVerification.ClassOnly = true`

### `EnableMain` — Use `main` function provided in application

false (default) | true

*This property affects Code Prover analysis only.*

Use `main` function provided in application, specified as true or false. If you set this property to false, the analysis generates a `main` function, if it is not present in the source files.

For more information, see `Verify whole application`.

Example: `opts.CodeProverVerification.EnableMain = true`

### `FunctionsCalledBeforeMain` — Functions that you want the generated main to call ahead of other functions

cell array of function names

*This property affects Code Prover analysis only.*

Functions that you want the generated main to call ahead of other functions, specified as a cell array of function names.

For more information, see `Initialization functions (-functions-called-before-main)`.

Example: `opts.CodeProverVerification.FunctionsCalledBeforeMain = {'func1','func2'}`

### `Main` — Use a Microsoft Visual C++ extensions of main

`'_tmain'` (default) | `'wmain'` | `'_tWinMain'` | `'wWinMain'` | `'WinMain'` | `'DllMain'`

*This property applies to a Code Prover analysis only .*

Use a Microsoft Visual C++ extension of main, specified as one of the predefined main extensions.

For more information, see `Main entry point (-main)`.

Example: `opts.CodeProverVerification.Main = 'wmain'`

### `MainGenerator` — Generate a main function if it is not present in source files

true (default) | false

*This property applies to a Code Prover analysis only .*

Generate a main function if it is not present in source files, specified as true or false.

For more information, see `Verify module or library (-main-generator)`.

Example: `opts.CodeProverVerification.MainGenerator = false`

### `MainGeneratorCalls` — Functions that you want the generated main to call after the initialization functions

`'unused'` (default) | `'none'` | `'all'` | `'custom=`*`function1[,function2,...]`*`'`

*This property applies to a Code Prover analysis only .*

Functions that you want the generated main to call after the initialization functions, specified as `'unused'`, `'all'`, `'none'`, or as a character array beginning with `custom=` followed by a comma-separated list of function names.

For more information, see `Functions to call (-main-generator-calls)`.

Example: `opts.CodeProverVerification.MainGeneratorCalls = 'all'`

### `MainGeneratorWriteVariables` — Global variables that you want the generated main to initialize

`'uninit'` (C++ default) | `'public'` (C default) | `'none'` | `'all'` |
`'custom=`*`variable1[,variable2,...]`*`'`

*This property applies to a Code Prover analysis only .*

Global variables that you want the generated main to initialize, specified as one of the predefined sets, or as a character array beginning with `custom=` followed by a comma-separated list of variable names.

For more information, see `Variables to initialize (-main-generator-writes-variables)`.

Example: `opts.CodeProverVerification.MainGeneratorWriteVariables = 'all'`

### `NoConstructorsInitCheck` — Do not check if class constructor initializes class members

false (default) | true

*This property applies to a Code Prover analysis only .*

Do not check if class constructor initializes class members, specified as true or false.

For more information, see `Skip member initialization check (-no-constructors-init-check)`.

Example: `opts.CodeProverVerification.NoConstructorsInitCheck = true`

### `UnitByUnit` — Verify each source file independently of other source files

false (default) | true

*This property affects Code Prover analysis only.*

Verify each source file independently of other source files, specified as true or false.

For more information, see `Verify files independently (-unit-by-unit)`.

Example: `opts.CodeProverVerification.UnitByUnit = true`

### `UnitByUnitCommonSource` — Files that you want to include with each source file during a file-by-file verification

cell array of file paths

*This property affects Code Prover analysis only.*

Files that you want to include with each source file during a file-by-file verification, specified as a cell array of file paths.

For more information, see `Common source files (-unit-by-unit-common-source)`.

Example: `opts.CodeProverVerification.UnitByUnitCommonSource = {'/inc/file1.h','/inc/file2.h'}`

**CodingRulesCodeMetrics**

### `AcAgcSubset` — Subset of MISRA AC AGC rules to check

`'OBL-rules'` (default) | `'OBL-REC-rules'` | `'single-unit-rules'` | `'system-decidable-rules'` | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA AC AGC rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA AC AGC (-misra-ac-agc)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA AC AGC rules, also set `EnableAcAgc` to true.

Example: `opts.CodingRulesCodeMetrics.AcAgcSubset = 'all-rules'`

Data Types: `char`

### `AllowedPragmas` — Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied

cell array of character vectors

Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied, specified as a cell array of character vectors. This property affects only MISRA C:2004 or MISRA AC AGC rule checking.

For more information, see `Allowed pragmas (-allowed-pragmas)`.

Example: `opts.CodingRulesCodeMetrics.AllowedPragmas = {'pragma_01','pragma_02'}`

Data Types: `cell`

### `AutosarCpp14` — Set of AUTOSAR C++ 14 rules to check

`'all'` (default) | `'required'` | `'automated'` | `polyspace.CodingRulesOptions` object | `'from-file'`

*This property affects Bug Finder only.*

Set of AUTOSAR C++ 14 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check AUTOSAR C++ 14 security checks (-autosar-cpp14)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for and Review Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check AUTOSAR C++ 14 rules, also set `EnableAutosarCpp14` to true.

Example: `opts.CodingRulesCodeMetrics.AutosarCpp14 = 'all'`

Data Types: `char`

**`BooleanTypes` — Data types the coding rule checker must treat as effectively Boolean**
cell array of character vectors

Data types that the coding rule checker must treat as effectively Boolean, specified as a cell array of character vectors.

For more information, see `Effective boolean types (-boolean-types)`.

Example: `opts.CodingRulesCodeMetrics.BooleanTypes = {'boolean1_t','boolean2_t'}`

Data Types: `cell`

**`CertC` — Set of CERT C rules and recommendations to check**
`'all'` (default) | `'publish-2016'` | `'all-rules'` | `polyspace.CodingRulesOptions` object | `'from-file'`

*This property affects Bug Finder only.*

Set of CERT C rules and recommendations to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C security checks (-cert-c)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `from-file` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for and Review Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C rules and recommendations, also set `EnableCertC` to true.

Example: `opts.CodingRulesCodeMetrics.CertC = 'all'`

Data Types: `char`

### `CertCpp` — Set of CERT C++ rules to check

`'all'` (default) | `polyspace.CodingRulesOptions` object | `'from-file'`

*This property affects Bug Finder only.*

Set of CERT C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C++ security checks (-cert-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C++ rules, also set `EnableCertCpp` to true.

Example: `opts.CodingRulesCodeMetrics.CertCpp = 'all'`

Data Types: `char`

### `CheckersSelectionByFile` — File that defines custom set of coding standard checkers

full file path of `.xml` file

File where you define a custom set of coding standards checkers to check, specified as a `.xml` file. You can, in the same file, define a custom set of checkers for each of the coding standards that Polyspace supports. To create a file that defines a custom selection of coding standard checkers, in the Polyspace interface, select a coding standard on the **Coding Standards & Code Metrics** node of the **Configuration** pane and click **Edit**.

For more information, see `Set checkers by file (-checkers-selection-file)`.

Example: `opts.CodingRulesCodeMetrics.CheckersSelectionByFile = 'C:\ps_settings \coding_rules\custom_rules.xml'`

Data Types: `char`

### `CodeMetrics` — Activate code metric calculations

false (default) | true

Activate code metric calculations, specified as true or false. If this property is turned off, Polyspace does not calculate code metrics even if you upload your results to Polyspace Metrics.

For more information about the code metrics, see `Calculate code metrics (-code-metrics)`.

If you assign a coding rules options object to this property, an XML file gets created automatically with the rules specified.

Example: `opts.CodingRulesCodeMetrics.CodeMetrics = true`

### `Cwe` — Set of CWE rules to check

`'all'` (default) | `'cwe-658-659'` | `'from-file'` | `polyspace.CodingRulesOptions` object

*This property affects Bug Finder only.*

Set of CWE rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CWE (-cwe)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C++ rules, also set `EnableCwe` to true.

Example: `opts.CodingRulesCodeMetrics.Cwe = 'cwe-658-659'`

Data Types: `char`

### `EnableAcAgc` — Check MISRA AC AGC rules

false (default) | true

Check MISRA AC AGC rules, specified as true or false. To customize which rules are checked, use `AcAgcSubset`.

For more information about the MISRA AC AGC checker, see `Check MISRA AC AGC (-misra-ac-agc)`.

Example: `opts.CodingRulesCodeMetrics.EnableAcAgc = true;`

### `EnableAutosarCpp14` — Check AUTOSAR C++ 14 rules

false (default) | true

*This property affects Bug Finder only.*

Check AUTOSAR C++ 14 rules, specified as true or false. To customize which rules are checked, use `AutosarCpp14`.

For more information about the AUTOSAR C++ 14 checker, see `Check AUTOSAR C++ 14 checks (-autosar-cpp14)`.

Example: `opts.CodingRulesCodeMetrics.EnableAutosarCpp14 = true;`

### `EnableCertC` — check CERT C rules and recommendations

false (default) | true

*This property affects Bug Finder only.*

Check CERT C rules and recommendations, specified as true or false. To customize which rules are checked, use `CertC`.

For more information about the CERT C checker, see `Check SEI CERT-C checks (-cert-c)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertC = true;`

### `EnableCertCpp` — check CERT C++ rules

false (default) | true

*This property affects Bug Finder only.*

Check CERT C++ rules, specified as true or false. To customize which rules are checked, use `CertCpp`.

For more information about the CERT C++ checker, see `Check SEI-CERT-C++ (-cert-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertCpp = true;`

### `EnableCheckersSelectionByFile` — Check custom set of coding standard checkers

false (default) | true

Check custom set of coding standard checkers, specified as true or false. If you set this property to true, specify the path of checkers selection XML file for the property `CheckersSelectionByFile` and set one of these properties to 'from-file' to enable the corresponding coding standard custom set:

- `opts.CodingRulesCodeMetrics.AcAgcSubset='from-file'`
- `opts.CodingRulesCodeMetrics.AutosarCpp14='from-file'`
- `opts.CodingRulesCodeMetrics.CertC='from-file'`
- `opts.CodingRulesCodeMetrics.CertCpp='from-file'`
- `opts.CodingRulesCodeMetrics.Cwe='from-file'`
- `opts.CodingRulesCodeMetrics.Guidelines='from-file'`
- `opts.CodingRulesCodeMetrics.Iso17961='from-file'`
- `opts.CodingRulesCodeMetrics.JsfSubset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraC3Subset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCSubset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCppSubset='from-file'`

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;`

### `EnableCustomRules` — Check custom coding rules

false (default) | true

Check custom coding rules, specified as true or false. The file you specify with `CheckersSelectionByFile` defines the custom coding rules.

Use with `EnableCheckersSelectionByFile`.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCustomRules = true;`

**EnableCwe — check CWE rules**
false (default) | true

*This property affects Bug Finder only.*

Check CWE rules, specified as true or false. To customize which rules are checked, use Cwe.

For more information about the CWE checker, see Check CWE (-cwe).

Example: `opts.CodingRulesCodeMetrics.Cwe = true;`

**EnableGuidelines — Check for violations of coding guidelines**
false (default) | true

*This property affects Bug Finder only.*

Specify whether to check for violations of Guidelines. To customize which rules are checked, use Guidelines.

For more information about the Guidelines checker, see Check guidelines (-guidelines).

Example: `opts.CodingRulesCodeMetrics.EnableGuidelines = true;`

**EnableIso17961 — check ISO-17961 rules**
false (default) | true

*This property affects Bug Finder only.*

Check ISO/IEC TS 17961 rules, specified as true or false. To customize which rules are checked, use Iso17961.

For more information about the ISO-17961 checker, see Check ISO-17961 security checks (-iso-17961).

Example: `opts.CodingRulesCodeMetrics.EnableIso17961 = true;`

**EnableJsf — Check JSF C++ rules**
false (default) | true

Check JSF C++ rules, specified as true or false. To customize which rules are checked, use JsfSubset.

For more information, see Check JSF C++ rules (-jsf-coding-rules).

Example: `opts.CodingRulesCodeMetrics.EnableJsf = true;`

**EnableMisraC — Check MISRA C:2004 rules**
false (default) | true

Check MISRA C:2004 rules, specified as true or false. To customize which rules are checked, use MisraCSubset.

For more information, see Check MISRA C:2004 (-misra2).

Example: `opts.CodingRulesCodeMetrics.EnableMisraC = true;`

**EnableMisraC3 — Check MISRA C:2012 rules**
false (default) | true

Check MISRA C:2012 rules, specified as true or false. To customize which rules are checked, use `MisraC3Subset`.

For more information about the MISRA C:2012 checker, see `Check MISRA C:2012 (-misra3)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC3 = true;`

#### `EnableMisraCpp` — Check MISRA C++:2008 rules

false (default) | true

Check MISRA C++:2008 rules, specified as true or false. To customize which rules are checked, use `MisraCppSubset`.

For more information about the MISRA C++:2008 checker, see `Check MISRA C++ rules (-misra-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraCpp = true;`

#### `Guidelines` — Set of Guidelines to check

`'all'` (default) | `'his'` | `'from-file'`

*This property affects Bug Finder only.*

Sets of Guideline rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check guidelines (-guidelines)`.
- An XML file specifying Guidelines checkers and their thresholds. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”.

Example: `opts.CodingRulesCodeMetrics.Guidelines = 'his';`

#### `Iso17961` — Set of ISO-17961 rules to check

`'all'` (default) | `'decidable'` | `polyspace.CodingRulesOptions` object | `'from-file'`

*This property affects Bug Finder only.*

Set of ISO/IEC TS 17961 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check ISO-17961 (-iso-17961)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check ISO/IEC TS 17961 rules, also set `EnableIso17961` to true.

Example: `opts.CodingRulesCodeMetrics.Iso17961 = 'all'`

Data Types: `char`

### `JsfSubset` — Subset of JSF C++ rules to check

`'shall-rules'` (default) | `'shall-will-rules'` | `'all-rules'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of JSF C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check JSF C++ rules (-jsf-coding-rules)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check JSF C++ rules, set `EnableJsf` to true.

Example: `opts.CodingRulesCodeMetrics.JsfSubset = 'all-rules'`

Data Types: `char`

### `Misra3AgcMode` — Use the MISRA C:2012 categories for automatically generated code

false (default) | true

Use the MISRA C:2012 categories for automatically generated code, specified as true or false.

For more information, see `Use generated code requirements (-misra3-agc-mode)`.

Example: `opts.CodingRulesCodeMetrics.Misra3AgcMode = true;`

### `MisraC3Subset` — Subset of MISRA C:2012 rules to check

`'mandatory-required'` (default) | `'mandatory'` | `'single-unit-rules'` | `'system-decidable-rules'` | `'all'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C:2012 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2012 (-misra3)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for and Review Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2012 rules, also set `EnableMisraC3` to true.

Example: `opts.CodingRulesCodeMetrics.MisraC3Subset = 'all'`

Data Types: `char`

### `MisraCSubset` — Subset of MISRA C:2004 rules to check

`'required-rules'` (default) | `'single-unit-rules'` | `'system-decidable-rules'` | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C:2004 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2004 (-misra2)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for and Review Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2004 rules, also set `EnableMisraC` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCSubset = 'all-rules'`

Data Types: `char`

### `MisraCppSubset` — Subset of MISRA C++ rules

`'required-rules'` (default) | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C++:2008 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C++ rules (-misra-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C++ rules, set `EnableMisraCpp` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCppSubset = 'all-rules'`

Data Types: `char`

#### EnvironmentSettings

### `Dos` — Consider that file paths are in MS-DOS style

true (default) | false

Consider that file paths are in MS-DOS style, specified as true or false.

For more information, see `Code from DOS or Windows file system (-dos)`.

Example: `opts.EnvironmentSettings.Dos = true;`

### `IncludeFolders` — Include folders needed for compilation

cell array of include folder paths

Include folders needed for compilation, specified as a cell array of the include folder paths.

To specify all subfolders of a folder, use folder path followed by `**`, for instance, `'C:\includes\**'`. The notation follows the syntax of the `dir` function. See also “Specify Multiple Source Files”.

For more information, see `-I`.

Example: `opts.EnvironmentSettings.IncludeFolders = {'/includes','/com1/inc'};`

Example: `opts.EnvironmentSettings.IncludeFolders = {'C:\project1\common\includes'};`

Data Types: `cell`

### `Includes` — Files to be #include-ed by each C file

cell array of files

Files to be `#include`-ed by each C source file in the analysis, specified by a cell array of files.

For more information, see `Include (-include)`.

Example: `opts.EnvironmentSettings.Includes = {'/inc/inc_file.h','/inc/inc_math.h'}`

### `NoExternC` — Ignore linking errors inside extern blocks

false (default) | true

Ignore linking errors inside extern blocks, specified as true or false.

For more information, see `Ignore link errors (-no-extern-c)`.

Example: `opts.EnvironmentSettings.NoExternC = false;`

### `PostPreProcessingCommand` — Command or script to run on source files after preprocessing

character vector

Command or script to run on source files after preprocessing, specified as a character vector of the command to run.

For more information, see `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Example: Linux — `opts.EnvironmentSettings.PostPreProcessingCommand = [pwd,'/replace_keyword.pl']`

Example: Windows — `opts.EnvironmentSettings.PostPreProcessingCommand = '"C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl"'`

### `StopWithCompileError` — Stop analysis if a file does not compile

false (default) | true

Stop analysis if a file does not compile, specified as true or false.

For more information, see `Stop analysis if a file does not compile (-stop-if-compile-error)`.

Example: `opts.EnvironmentSettings.StopWithCompileError = true;`

#### InputsStubbing

### `DataRangeSpecifications` — Constrain global variables, function inputs, and return values of stubbed functions

file path

Constrain global variables, function inputs, and return values of stubbed functions specified by the path to an XML constraint file. For more information about the constraint file, see “Specify External Constraints for Polyspace Analysis”.

For more information about this option, see `Constraint setup (-data-range-specifications)`.

Example: `opts.InputsStubbing.DataRangeSpecifications = 'C:\project\constraint_file.xml'`

### `DoNotGenerateResultsFor` — Files on which you do not want analysis results

`'include-folders'` (default) | `'all-headers'` | `'custom=`*`folder1[,folder2,...]`*`'`

Files on which you do not want analysis results, specified by `'include-folders'`, `'all-headers'`, or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.GenerateResultsFor`. For more information, see `Do not generate results for (-do-not-generate-results-for)`.

Example: `opts.InputsStubbing.DoNotGenerateResultsFor = 'custom=C:\project\file1.c,C:\project\file2.c'`

### `GenerateResultsFor` — Files on which you want analysis results

`'source-headers'` (default) | `'all-headers'` | `'custom=`*`folder1[,folder2,...]`*`'`

Files on which you want analysis results, specified by `'source-headers'`, `'all-headers'`, or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.DoNotGenerateResultsFor`. For more information, see `Generate results for sources and (-generate-results-for)`.

Example: `opts.InputsStubbing.GenerateResultsFor = 'custom=C:\project\includes_common_1,C:\project\includes_common_2'`

**`FunctionsToStub` — Functions to stub during analysis**
cell array of function names

*This property affects Code Prover analysis only.*

Functions to stub during analysis, specified as a cell array of function names.

For more information, see `Functions to stub (-functions-to-stub)`.

Example: `opts.InputsStubbing.FunctionsToStub = {'func1', 'func2'}`

**`NoDefInitGlob` — Consider global variables as uninitialized**
false (default) | true

*This property affects Code Prover analysis only.*

Consider global variables as uninitialized, specified as true or false.

For more information, see `Ignore default initialization of global variables (-no-def-init-glob)`.

Example: `opts.InputsStubbing.NoDefInitGlob = true`

**`StubECoderLookupTables` — Specify that the analysis must stub functions in the generated code that use lookup tables**
true (default) | false

*This property applies only to a Code Prover analysis of code generated from models.*

Specify that the analysis must stub functions in the generated code that use lookup tables. By replacing the functions with stubs, the analysis assumes more precise return values for the functions.

For more information, see `Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)`.

Example: `opts.InputsStubbing.StubECoderLookupTables = true`

#### Macros

**`DefinedMacros` — Macros to be replaced**
cell array of macros

In preprocessed code, macros are replaced by the definition, specified in a cell array of macros and definitions. Specify the macro as `Macro=Value`. If you want Polyspace to ignore the macro, leave the `Value` blank. A macro with no equal sign replaces all instances of that macro by 1.

For more information, see `Preprocessor definitions (-D)`.

Example: `opts.Macros.DefinedMacros = {'uint32=int','name3=','var'}`

### `UndefinedMacros` — Macros to undefine

cell array of macros

In preprocessed code, macros are undefined, specified by a cell array of macros to undefine.

For more information, see `Disabled preprocessor definitions (-U)`.

Example: `opts.Macros.DefinedMacros = {'name1','name2'}`

#### MergedComputingSettings

### `BatchBugFinder` — Send Bug Finder analysis to remote server

false (default) | true

*This property affects Bug Finder analysis only.*

Send Bug Finder analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see:

- `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`
- “Run Analysis on Server”

Example: `opts.MergedComputingSettings.BatchBugFinder = true;`

### `BatchCodeProver` — Send Code Prover analysis to remote server

false (default) | true

*This property affects Code Prover analysis only.*

Send Code Prover analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see:

- `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`
- “Run Analysis on Server”

Example: `opts.MergedComputingSettings.BatchCodeProver = true;`

### `FastAnalysis` — Run Bug Finder analysis using faster local mode

false (default) | true

*This property affects Bug Finder analysis only.*

Use fast analysis mode for Bug Finder analysis, specified as true or false.

For more information, see `Use fast analysis mode for Bug Finder (-fast-analysis)`.

Example: `opts.MergedComputingSettings.FastAnalysis = true;`

#### MergedReporting

### `EnableReportGeneration` — Generate a report after the analysis

false (default) | true

After the analysis, generate a report, specified as true or false.

For more information, see `Generate report`.

Example: `opts.MergedReporting.EnableReportGeneration = true`

**`ReportOutputFormat` — Output format of generated report**
`'Word'` (default) | `'HTML'` | `'PDF'`

Output format of generated report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Output format (-report-output-format)`.

Example: `opts.MergedReporting.ReportOutputFormat = 'PDF'`

**`BugFinderReportTemplate` — Template for generating Bug Finder analysis report**
`'BugFinderSummary'` (default) | `'BugFinder'` | `'SecurityCWE'` | `'CodeMetrics'` | `'CodingStandards'`

*This property affects a Bug Finder analysis only.*

Template for generating analysis report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.BugFinderReportTemplate = 'CodeMetrics'`

**`CodeProverReportTemplate` — Template for generating Code Prover analysis report**
`'Developer'` (default) | `'CallHierarchy'` | `'CodeMetrics'` | `'CodingStandards'` | `'DeveloperReview'` | `'Developer_withGreenChecks'` | `'Quality'` | `'VariableAccess'`

*This property affects a Code Prover analysis only.*

Template for generating analysis report, specified as one of the predefined report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.CodeProverReportTemplate = 'CodeMetrics'`

#### Multitasking

**`ArxmlMultitasking` — Specify path of ARXML files to parse for multitasking configuration**
cell array of file paths

Specify the path to the ARXML files the software parses to set up your multitasking configuration.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `autosar`.

For more information, see `ARXML files selection (-autosar-multitasking)`

Example: `opts.Multitasking.ArxmlMultitasking={'C:\Polyspace_Workspace\AUTOSAR\myFile.arxml'}`

### `CriticalSectionBegin` — Functions that begin critical sections

cell array of critical section function names

Functions that begin critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionEnd`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionBegin = {'function1:cs1','function2:cs2'}`

### `CriticalSectionEnd` — Functions that end critical sections

cell array of critical section function names

Functions that end critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionBegin`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionEnd = {'function1:cs1','function2:cs2'}`

### `CyclicTasks` — Specify functions that represent cyclic tasks

cell array of function names

Specify functions that represent cyclic tasks.

To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Cyclic tasks (-cyclic-tasks)`.

Example: `opts.Multitasking.CyclicTasks = {'function1','function2'}`

### `EnableConcurrencyDetection` — Enable automatic detection of certain families of threading functions

false (default) | true

*This property affects Code Prover analysis only.*

Enable automatic detection of certain families of threading functions, specified as true or false.

For more information, see `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

Example: `opts.Multitasking.EnableConcurrencyDetection = true`

### `EnableExternalMultitasking` — Enable automatic multitasking configuration from external file definitions

false (default) | true

Enable multitasking configuration of your projects from external files you provide. Configure multitasking from ARXML files for an AUTOSAR project, or from OIL files for an OSEK project.

Activate this option to enable `Multitasking.ArxmlMultitasking` or `Multitasking.OsekMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.EnableExternalMultitasking = 1`

**`EnableMultitasking` — Configure multitasking manually**
false (default) | true

Configure multitasking manually by specifying `true`. This property activates the other manual, multitasking properties.

For more information, see `Configure multitasking manually`.

Example: `opts.Multitasking.EnableMultitasking = 1`

**`EntryPoints` — Functions that serve as entry-points to your multitasking application**
cell array of entry-point function names

Functions that serve as entry-points to your multitasking application specified as a cell array of entry-point function names. To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Tasks (-entry-points)`.

Example: `opts.Multitasking.EntryPoints = {'function1','function2'}`

**`ExternalMultitaskingType` — Specify type of file to parse for multitasking configuration**
`'osek'` (default) | `'autosar'`

Specify the type of file the software parses to set up your multitasking configuration:

- For `osek` type, the analysis looks for OIL files in the file or folder paths that you specify.
- For `autosar` type, the analysis looks for ARXML files in the file paths that you specify.

To activate this option, specify `Multitasking.EnableExternalMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.ExternalMultitaskingType = 'autosar'`

**`Interrupts` — Specify functions that represent nonpreemptable interrupts**
cell array of function names

Specify functions that represent nonpreemptable interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Interrupts (-interrupts)`.

Example: `opts.Multitasking.Interrupts = {'function1','function2'}`

**`InterruptsDisableAll` — Specify routine that disable interrupts**
cell array with one function name

*This property affects Bug Finder analysis only.*

Specify function that disables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsDisableAll = {'function'}`

### `InterruptsEnableAll` — Specify routine that reenable interrupts

cell array with one function name

*This property affects Bug Finder analysis only.*

Specify function that reenables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsEnableAll = {'function'}`

### `OsekMultitasking` — Specify path of OIL files to parse for multitasking configuration

`'auto'` (default) | `'custom=`*`folder1[,folder2,...]`*`'`

Specify the path to the OIL files the software parses to set up your multitasking configuration:

- In the mode specified with `'auto'`, the analysis uses OIL files in your project source and include folders, but not their subfolders.
- In the mode specified with `'custom=`*`folder1[,folder2,...]`*`'`, the analysis uses the OIL files at the specified path, and the path subfolders.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `osek`.

For more information, see `OIL files selection (-osek-multitasking)`

Example: `opts.Multitasking.OsekMultitasking = 'custom=file_path, dir_path'`

### `TemporalExclusion` — Entry-point functions that cannot execute concurrently

cell array of entry-point function names

Entry-point functions that cannot execute concurrently specified as a cell array of entry-point function names. Each set of exclusive tasks is one cell array entry with functions separated by spaces. To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Temporally exclusive tasks (-temporal-exclusions-file)`.

Example: `opts.Multitasking.TemporalExclusion = {'function1 function2', 'function3 function4 function5'}` where function1 and function2 are temporally exclusive, and function3, function4, and function 5 are temporally exclusive.

**Precision (Affects Code Prover Only)**

**`ContextSensitivity` — Store call context information to identify function call that caused errors**
`'none'` (default) | `'auto'` | `'custom=`*`function1[,function2,...]`*`'`

*This property affects Code Prover analysis only.*

Store call context information to identify a function call that caused errors, specified as `none`, `auto`, or as a character array beginning with `custom=` followed by a list of comma-separated function names.

For more information, see `Sensitivity context (-context-sensitivity)`.

Example: `opts.Precision.ContextSensitivity = 'auto'`

Example: `opts.Precision.ContextSensitivity = 'custom=func1'`

**`ModulesPrecision` — Source files you want to verify at higher precision**
cell array of file names and precision levels

*This property affects Code Prover analysis only.*

Source files that you want to verify at higher precision, specified as a cell array of file names without the extension and precision levels using this syntax: *`filename`*`:O`*`level`*

For more information, see `Specific precision (-modules-precision)`.

Example: `opts.Precision.ModulesPrecision = {'file1:O0', 'file2:O3'}`

**`OLevel` — Precision level for the verification**
2 (default) | 0 | 1 | 3

*This property affects Code Prover analysis only.*

Precision level for the verification, specified as 0, 1, 2, or 3.

For more information, see `Precision level (-O)`.

Example: `opts.Precision.OLevel = 3`

**`PathSensitivityDelta` — Avoid certain verification approximations for code with fewer lines**
positive integer

*This property affects Code Prover analysis only.*

Avoid certain verification approximations for code with fewer lines, specified as a positive integer representing how sensitive the analysis is. Higher values can increase verification time exponentially.

For more information, see `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

Example: `opts.Precision.PathSensitivityDelta = 2`

**`Timeout` — Time limit on your verification**
character vector

*This property affects Code Prover analysis only.*

Time limit on your verification, specified as a character vector of time in hours.

For more information, see `Verification time limit (-timeout)`.

Example: `opts.Precision.Timeout = '5.75'`

### `To` — Number of times the verification process runs

`'Software Safety Analysis level 2'` (default) | `'Software Safety Analysis level 0'` | `'Software Safety Analysis level 1'` | `'Software Safety Analysis level 3'` | `'Software Safety Analysis level 4'` | `'Source Compliance Checking'` | `'other'`

*This property affects Code Prover analysis only.*

Number of times the verification process runs, specified as one of the preset analysis levels.

For more information, see `Verification level (-to)`.

Example: `opts.Precision.To = 'Software Safety Analysis level 3'`

#### Scaling (Affects Code Prover Only)

### `Inline` — Functions on which separate results must be generated for each function call

cell array of function names

*This property affects Code Prover analysis only.*

Functions on which separate results must be generated for each function call, specified as a cell array of function names.

For more information, see `Inline (-inline)`.

Example: `opts.Scaling.Inline = {'func1','func2'}`

### `KLimiting` — Limit depth of analysis for nested structures

positive integer

*This property affects Code Prover analysis only.*

Limit depth of analysis for nested structures, specified as a positive integer indicating how many levels into a nested structure to verify.

For more information, see `Depth of verification inside structures (-k-limiting)`.

Example: `opts.Scaling.KLimiting = 3`

#### TargetCompiler

### `Compiler` — Compiler that builds your source code

`'generic'` (default) | `'gnu3.4'` | `'gnu4.6'` | `'gnu4.7'` | `'gnu4.8'` | `'gnu4.9'` | `'gnu5.x'` | `'gnu6.x'` | `'gnu7.x'` | `'clang3.x'` | `'clang4.x'` | `'clang5.x'` | `'visual9.0'` | `'visual10'` | `'visual11.0'` | `'visual12.0'` | `'visual14.0'` | `'visual15.x'` | `'keil'` | `'iar'` | `'armcc'` | `'armclang'` | `'codewarrior'` | `'diab'` | `'greenhills'` | `'iar-ew'` | `'renesas'` | `'tasking'` | `'ti'`

Compiler that builds your source code.

For more information, see `Compiler (-compiler)`.

Example: `opts.TargetCompiler.Compiler = 'Visual11.0'`

### `CppVersion` — Specify C++ standard version followed in code

`'defined-by-compiler'` (default) | `'cpp03'` | `'cpp11'` | `'cpp14'` | `'cpp17'`

Specify C++ standard version followed in code, specified as a character vector.

For more information, see `C++ standard version (-cpp-version)`.

Example: `opts.TargetCompiler.CppVersion = 'cpp11';`

### `CVersion` — Specify C standard version followed in code

`'defined-by-compiler'` (default) | `'c90'` | `'c99'` | `'c11'`

Specify C standard version followed in code, specified as a character vector.

For more information, see `C standard version (-c-version)`.

Example: `opts.TargetCompiler.CVersion = 'c90';`

### `DivRoundDown` — Round down quotients from division or modulus of negative numbers

false (default) | true

Round down quotients from division or modulus of negative numbers, specified as true or false.

For more information, see `Division round down (-div-round-down)`.

Example: `opts.TargetCompiler.DivRoundDown = true`

### `EnumTypeDefinition` — Base type representation of enum

`'defined-by-compiler'` (default) | `'auto-signed-first'` | `'auto-unsigned-first'`

Base type representation of enum, specified by an allowed base-type set. For more information about the different values, see `Enum type definition (-enum-type-definition)`.

Example: `opts.TargetCompiler.EnumTypeDefinition = 'auto-unsigned-first'`

### `IgnorePragmaPack` — Ignore #pragma pack directives

false (default) | true

Ignore #pragma pack directives, specified as true or false.

For more information, see `Ignore pragma pack directives (-ignore-pragma-pack)`.

Example: `opts.TargetCompiler.IgnorePragmaPack = true`

### `Language` — Language of analysis

`'C-CPP'` (default) | `'C'` | `'CPP'`

This property is read-only.

Language of the analysis, specified during the object construction. This value changes which properties appear.

For more information, see `Source code language (-lang)`.

**LogicalSignedRightShift — Treatment of signed bit on signed variables**
'Arithmetical' (default) | 'Logical'

Treatment of signed bit on signed variables, specified as Arithmetical or Logical. For more information, see Signed right shift (-logical-signed-right-shift).

Example: opts.TargetCompiler.LogicalSignedRightShift = 'Logical'

**NoUliterals — Do not use predefined typedefs for char16_t or char32_t**
false (default) | true

Do not use predefined typedefs for char16_t or char32_t, specified as true or false. For more information, see Block char16/32_t types (-no-uliterals).

Example: opts.TargetCompiler.NoUliterals = true

**PackAlignmentValue — Default structure packing alignment**
'defined-by-compiler' (default) | '1' | '2' | '4' | '8' | '16'

Default structure packing alignment, specified as 'defined-by-compiler', '1', '2', '4', '8', or '16'. This property is available only for Visual C++ code.

For more information, see Pack alignment value (-pack-alignment-value).

Example: opts.TargetCompiler.PackAlignmentValue = '4'

**SfrTypes — sfr types**
cell array of sfr keywords

sfr types, specified as a cell array of sfr keywords using the syntax *sfr_name=size_in_bits*. For more information, see Sfr type support (-sfr-types).

This option only applies when you set TargetCompiler.Compiler to keil or iar.

Example: opts.TargetCompiler.SfrTypes = {'sfr32=32'}

**SizeTTypeIs — Underlying type of size_t**
'defined-by-compiler' (default) | 'unsigned-int' | 'unsigned-long' | 'unsigned-long-long'

Underlying type of size_t, specified as 'defined-by-compiler', 'unsigned-int', 'unsigned-long', or 'unsigned-long-long'. See Management of size_t (-size-t-type-is).

Example: opts.TargetCompiler.SizeTTypeIs = 'unsigned-long'

**Target — Target processor**
'i386' (default) | 'arm' | 'arm64' | 'avr' | 'c-167' | 'c166' | 'c18' | 'c28x' | 'c6000' | 'coldfire' | 'hc08' | 'hc12' | 'm68k' | 'mcore' | 'mips' | 'mpc5xx' | 'msp430' | 'necv850' | 'powerpc' | 'powerpc64' | 'rh850' | 'rl78' | 'rx' | 's12z' | 'sharc21x61' | 'sparc' | 'superh' | 'tms320c3x' | 'tricore' | 'x86_64' | generic target object

Set size of data types and endianness of processor, specified as one of the predefined target processors or a generic target object.

For more information about the predefined processors, see Target processor type (-target).

For more information about creating a generic target, see polyspace.GenericTargetOptions.

Example: `opts.TargetCompiler.Target = 'hc12'`

**`WcharTTypeIs` — Underlying type of `wchar_t`**
`'defined-by-compiler'` (default) | `'signed-short'` | `'unsigned-short'` | `'signed-int'` | `'unsigned-int'` | `'signed-long'` | `'unsigned-long'`

Underlying type of `wchar_t`, specified as `'defined-by-compiler'`, `'signed-short'`, `'unsigned-short'`, `'signed-int'`, `'unsigned-int'`, `'signed-long'`, or `'unsigned-long'`. See `Management of wchar_t (-wchar-t-type-is)`.

Example: `opts.TargetCompiler.WcharTTypeIs = 'unsigned-int'`

**VerificationAssumption (Affects Code Prover Only)**

**`ConsiderVolatileQualifierOnFields` — Assume that volatile qualified structure fields can have all possible values at any point in code**
false (default) | true

*This property affects Code Prover analysis only.*

Assume that volatile qualified structure fields can have all possible values at any point in code.

For more information, see `Consider volatile qualifier on fields (-consider-volatile-qualifier-on-fields)`.

Example: `opts.VerificationAssumption.ConsiderVolatileQualifierOnFields = true`

**`ConstraintPointersMayBeNull` — Specify that environment pointers can be NULL unless constrained otherwise**
false (default) | true

*This property affects Code Prover analysis only.*

Specify that environment pointers can be NULL unless constrained otherwise.

For more information, see `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`.

Example: `opts.VerificationAssumption.ConstraintPointersMayBeNull = true`

**`FloatRoundingMode` — Rounding modes to consider when determining the results of floating-point arithmetic**
`to-nearest` (default) | `all`

*This property affects Code Prover analysis only.*

Rounding modes to consider when determining the results of floating-point arithmetic, specified as `to-nearest` or `all`.

For more information, see `Float rounding mode (-float-rounding-mode)`.

Example: `opts.VerificationAssumption.FloatRoundingMode = 'all'`

**Other Properties**

**`Author` — Project author**
username of current user (default) | character vector

Name of project author, specified as a character vector.

For more information, see `-author`.

Example: `opts.Author = 'JaneDoe'`

### `ImportComments` — Import comments and justifications from previous analysis

character vector

To import comments and justifications from a previous analysis, specify the path to the results folder of the previous analysis.

You can also point to a previous results folder to see only new results compared to the previous run. See “Compare Results from Different Polyspace Runs by Using MATLAB Scripts”.

For more information, see `-import-comments`

Example: `opts.ImportComments = fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example','Module_1','BF_Result')`

### `Prog` — Project name

`PolyspaceProject` (default) | character vector

Project name, specified as a character vector.

For more information, see `-prog`.

Example: `opts.Prog = 'myProject'`

### `ResultsDir` — Location to store results

folder path

Location to store results, specified as a folder path. By default, the results are stored in the current folder.

For more information, see `-results-dir`.

You can also create a separate results folder for each new run. See “Compare Results from Different Polyspace Runs by Using MATLAB Scripts”.

Example: `opts.ResultsDir = 'C:\project\myproject\results\'`

### `Sources` — Source files

cell array of files

Source files to analyze, specified as a cell array of files.

To specify all files in a folder, use folder path followed by `*`, for instance, `'C:\src\*'`. To specify all files in a folder and its subfolders, use folder path followed by `**`, for instance, `'C:\src\**'`. The notation follows the syntax of the `dir` function. See also “Specify Multiple Source Files”.

For more information, see `-sources`.

Example: `opts.Sources = {'file1.c', 'file2.c', 'file3.c'}`

Example: `opts.Sources = {'project/src1/file1.c', 'project/src2/file2.c', 'project/src3/file3.c'}`

### `Version` — Project version number

`'1.0'` (default) | character array of a number

Version number of project, specified as a character array of a number. This option is useful if you upload your results to Polyspace Metrics. If you increment version numbers each time that you reanalyze your object, you can compare the results from two versions in Polyspace Metrics.

For more information, see `-v[ersion]`.

Example: `opts.Version = '2.3'`

## Version History

**Introduced in R2017a**

### R2023a: `CheckersPreset` option value `CWE` is removed

*Not recommended starting in R2023a*

The option to select the CWE subset of Polyspace Bug Finder defect checkers is removed. To check your code for compliance with the CWE standard, use these properties instead:

```
opts=polyspace.Options;
opts.CodingRulesCodeMetrics.EnableCwe=1
opts.CodingRulesCodeMetrics.Cwe='all';
```

See also `Check CWE (-cwe)`.

### R2021b: Automatic Orange Tester is removed

*Not recommended starting in R2021b*

The Automatic Orange Tester is removed. If you use these properties in your scripts, remove them (`opts=polyspace.Options('C')`):

- `opts.Advanced.AutomaticOrangeTester`
- `opts.Advanced.AutomaticOrangeTesterLoopMaxIteration`
- `opts.Advanced.AutomaticOrangeTesterTestsNumber`
- `opts.Advanced.AutomaticOrangeTesterTimeout`

## See Also

**Topics**

"Complete List of Polyspace Bug Finder Analysis Engine Options"

# polyspace.ModelLinkOptions Properties

Customize Polyspace analysis of generated code with options object properties

## Description

To customize your Polyspace analysis of generated code, modify the `polyspace.ModelLinkOptions` object properties. Each property corresponds to an analysis option on the **Configuration** pane in the Polyspace user interface.

The properties are grouped using the same categories as the **Configuration** pane. This page only shows what values each property can take. For details about:

- The different options, see the analysis options reference pages.
- How to create and use the object, see `polyspace.ModelLinkOptions`.

  The same properties are also available with the deprecated classes `polyspace.ModelLinkBugFinderOptions` and `polyspace.ModelLinkCodeProverOptions`.

Each property description below also highlights if the option affects only one of Bug Finder or Code Prover.

**Note** Some options might not be available depending on the language setting of the object. You can set the source code language (`Language`) to `'C'`, `'CPP'` or `'C-CPP'` during object creation, but cannot change it later.

### Properties

**Advanced**

**`Additional` — Additional flags for analysis**
character vector

Additional flags for analysis specified as a character vector.

For more information, see `Other`.

Example: `opts.Advanced.Additional = '-extra-flags -option -extra-flags value'`

**`PostAnalysisCommand` — Command or script software should execute after analysis finishes**
character vector

Command or script software should execute after analysis finishes, specified as a character vector.

For more information, see `Command/script to apply after the end of the code verification (-post-analysis-command)`.

Example: `opts.Advanced.PostAnalysisCommand = '"C:\Program Files\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"'`

#### BugFinderAnalysis (Affects Bug Finder Only)

### `CheckersList` — List of custom checkers to activate

`polyspace.DefectsOptions` object | cell array of defect acronyms

*This property affects Bug Finder analysis only.*

List of custom checkers to activate specified by using the name of a `polyspace.DefectsOptions` object or a cell array of defect acronyms. To use this custom list in your analysis, set `CheckersPreset` to `custom`.

For more information, see `polyspace.DefectsOptionspolyspace.DefectsOptions`.

Example: `defects = polyspace.DefectsOptions; opts.BugFinderAnalysis.CheckersList = defects`

Example: `opts.BugFinderAnalysis.CheckersList = {'INT_ZERO_DIV','FLOAT_ZERO_DIV'}`

### `CheckersPreset` — Subset of Bug Finder defects

`'default'` (default) | `'all'` | `'custom'`

*This property affects Bug Finder analysis only.*

Preset checker list, specified as a character vector of one of the preset options: `'default'`, `'all'`,or `'custom'`. To use `'custom'`, specify a value for the property `BugFinderAnalysis.CheckersList`.

For more information, see `Find defects (-checkers)`.

Example: `opts.BugFinderAnalysis.CheckersPreset = 'all'`

### `ChecksUsingSystemInputValues` — Activate stricter checks for system inputs

false (default) | true

*This property affects Bug Finder analysis only.*

Activate stricter checks that consider all possible value for:

- Global variables.
- Reads of volatile variables.
- Returns of stubbed functions.
- Inputs to functions specified with **SystemInputsFrom**.

The analysis considers all possible values for a subset of **Numerical** and **Static memory** defects.

This property is equivalent to the **Run stricter checks considering all values of system inputs** check box in the Polyspace interface.

For more information, see `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`.

Example: `opts.BugFinderAnalysis.ChecksUsingSystemInputValues = true`

### `EnableCheckers` — Activate defect checking

true (default) | false

*This property affects Bug Finder analysis only.*

Activate defect checking, specified as true or false. Setting this property to false disables all defects. If you want to disable defect checking but still get results, turn on coding rules checking or code metric checking.

This property is equivalent to the **Find defects** check box in the Polyspace interface.

Example: `opts.BugFinderAnalysis.EnableCheckers = false`

### `SystemInputsFrom` — List of functions for which you run stricter checks

`'auto'` (default) | `'uncalled'` | `'all'` | `'custom'`

*This property affects Bug Finder analysis only.*

Functions for which you want to run stricter checks that consider all possible values of the function inputs. Specify the list of functions as `'auto'`, `'uncalled'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of function names.

To enable this option, set `BugFinderAnalysis.ChecksUsingSystemInputValues = true`.

For more information, see `Consider inputs to these functions (-system-inputs-from)`.

Example: `opts.BugFinderAnalysis.SystemInputsFrom = 'custom=foo,bar'`

#### ChecksAssumption (Affects Code Prover Only)

### `AllowNegativeOperandInShift` — Allow left shift operations on a negative number

true (default) | false

*This property affects Code Prover analysis only.*

Allow left shift operations on a negative number, specified as true or false.

For more information, see `Allow negative operand for left shifts (-allow-negative-operand-in-shift)`.

Example: `opts.ChecksAssumption.AllowNegativeOperandInShift = true`

### `AllowNonFiniteFloats` — Incorporate infinities and/or NaNs

false (default) | true

*This property affects Code Prover analysis only.*

Incorporate infinities and/or NaNs, specified as true or false.

For more information, see `Consider non finite floats (-allow-non-finite-floats)`.

Example: `opts.ChecksAssumption.AllowNonFiniteFloats = true`

### `AllowPtrArithOnStruct` — Allow arithmetic on pointer to a structure field so that it points to another field

false (default) | true

*This property affects Code Prover analysis only.*

Allow arithmetic on pointer to a structure field so that it points to another field, specified as true or false.

For more information, see `Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)`.

Example: `opts.ChecksAssumption.AllowPtrArithOnStruct = true`

**`CheckInfinite` — Detect floating-point operations that result in infinities**
`'allow'` (default) | `'warn-first'` | `'forbid'`

*This property affects Code Prover analysis only.*

Detect floating-point operations that result in infinities.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `Infinities (-check-infinite)`.

Example: `opts.ChecksAssumption.CheckInfinite = 'forbid'`

**`CheckNan` — Detect floating-point operations that result in NaN-s**
`'allow'` (default) | `'warn-first'` | `'forbid'`

*This property affects Code Prover analysis only.*

Detect floating-point operations that result in NaN-s.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `NaNs (-check-nan)`.

Example: `opts.ChecksAssumption.CheckNan = 'forbid'`

**`CheckSubnormal` — Detect operations that result in subnormal floating point values**
`'allow'` (default) | `'warn-first'` | `'warn-all'` | `'forbid'`

*This property affects Code Prover analysis only.*

Detect operations that result in subnormal floating point values.

For more information, see `Subnormal detection mode (-check-subnormal)`.

Example: `opts.ChecksAssumption.CheckSubnormal = 'forbid'`

**`DetectPointerEscape` — Find cases where a function returns a pointer to one of its local variables**
false (default) | true

*This property affects Code Prover analysis only.*

Find cases where a function returns a pointer to one of its local variables, specified as true or false.

For more information, see `Detect stack pointer dereference outside scope (-detect-pointer-escape)`.

Example: `opts.ChecksAssumption.DetectPointerEscape = true`

**`DisableInitializationChecks` — Disable checks for noninitialized variables and pointers**
false (default) | true

*This property affects Code Prover analysis only.*

Disable checks for noninitialized variables and pointers, specified as true or false.

For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

Example: `opts.ChecksAssumption.DisableInitializationChecks = true`

### `PermissiveFunctionPointer` — Allow type mismatch between function pointers and the functions they point to

false (default) | true

*This property affects Code Prover analysis only.*

Allow type mismatch between function pointers and the functions they point to, specified as true or false.

For more information, see `Permissive function pointer calls (-permissive-function-pointer)`.

Example: `opts.ChecksAssumption.PermissiveFunctionPointer = true`

### `SignedIntegerOverflows` — Behavior of signed integer overflows

`'warn-with-wrap-around'` (default) | `'forbid'` | `'allow'`

*This property affects Code Prover analysis only.*

Enable the check for signed integer overflows and the assumptions to make following an overflow specified as `'forbid'`, `'allow'`, or `'warn-with-wrap-around'`.

For more information, see `Overflow mode for signed integer (-signed-integer-overflows)`.

Example: `opts.ChecksAssumption.SignedIntegerOverflows = 'warn-with-wrap-around'`

### `SizeInBytes` — Allow a pointer with insufficient memory buffer to point to a structure

false (default) | true

*This property affects Code Prover analysis only.*

Allow a pointer with insufficient memory buffer to point to a structure, specified as true or false.

For more information, see `Allow incomplete or partial allocation of structures (-size-in-bytes)`.

Example: `opts.ChecksAssumption.SizeInBytes = true`

### `UncalledFunctionCheck` — Detect functions that are not called directly or indirectly from main or another entry-point function

`'none'` (default) | `'never-called'` | `'called-from-unreachable'` | `'all'`

*This property affects Code Prover analysis only.*

Detect functions that are not called directly or indirectly from main or another entry-point function, specified as `'none'`, `'never-called'`, `'called-from-unreachable'`, or `'all'`.

For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

Example: `opts.ChecksAssumption.UncalledFunctionCheck = 'all'`

### `UnsignedIntegerOverflows` — Behavior of unsigned integer overflows

`'allow'` (default) | `'forbid'` | `'warn-with-wrap-around'`

*This property affects Code Prover analysis only.*

Enable the check for unsigned integer overflows and the assumptions to make following an overflow, specified as `'forbid'`, `'allow'`, or `'warn-with-wrap-around'`.

For more information, see `Overflow mode for unsigned integer (-unsigned-integer-overflows)`.

Example: `opts.ChecksAssumption.UnsignedIntegerOverflows = 'allow'`

#### CodeProverVerification (Affects Code Prover only)

### `ClassAnalyzer` — Classes that you want to verify

`'none'` (default) | `'all'` | `'custom=`*`class1[,class2,...]`*`'`

*This property affects Code Prover analysis only.*

Classes that you want to verify, specified as `'all'`, `'none'`, or as a character array beginning with `custom=` followed by a comma-separated list of class names.

For more information, see `Class (-class-analyzer)`.

Example: `opts.CodeProverVerification.ClassAnalyzer = 'none'`

### `FunctionsCalledAfterLoop` — Functions that the generated main must call after the cyclic code loop

cell array of function names

*This property affects Code Prover analysis only.*

Functions that the generated main must call after the cyclic code loop, specified as a cell array of function names.

For more information, see `Termination functions (-functions-called-after-loop)`.

Example: `opts.CodeProverVerification.FunctionsCalledAfterLoop = {'func1','func2'}`

### `FunctionsCalledBeforeLoop` — Functions that the generated main must call before the cyclic code loop

cell array of function names

*This property affects Code Prover analysis only.*

Model Link only. Functions that the generated main must call before the cyclic code loop, specified as a cell array of function names.

For more information, see `Initialization functions (-functions-called-before-loop))`.

Example: `opts.CodeProverVerification.FunctionsCalledBeforeLoop = {'func1','func2'}`

### `FunctionsCalledInLoop` — Functions that the generated main must call in the cyclic code loop

`'none'` (default) | `'all'` | `'custom=`*`function1[,function2,...]`*`'`

*This property affects Code Prover analysis only.*

Functions that the generated main must call in the cyclic code loop, specified as `'none'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of function names..

For more information, see `Step functions (-functions-called-in-loop)`.

Example: `opts.CodeProverVerification.FunctionsCalledInLoop = 'all'`

### `MainGenerator` — Generate a main function if it is not present in source files

true (default) | false

*This property affects Code Prover analysis only.*

Generate a main function if it is not present in source files, specified as true or false.

For more information, see `Verify module or library (-main-generator)`.

Example: `opts.CodeProverVerification.MainGenerator = false`

### `VariablesWrittenBeforeLoop` — Variables that the generated main must initialize before the cyclic code loop

`'none'` (default) | `'all'` | `'custom=`*`variable1[,variable2,...]`*`'`

*This property affects Code Prover analysis only.*

Variables that the generated main must initialize before the cyclic code loop, specified as `'none'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of variable names.

For more information, see `Parameters (-variables-written-before-loop)`.

Example: `opts.CodeProverVerification.VariablesWrittenBeforeLoop = 'all'`

### `VariablesWrittenInLoop` — Variables that the generated main must initialize in the cyclic code loop

`'none'` (default) | `'all'` | `'custom=`*`variable1[,variable2,...]`*`'`

*This property affects Code Prover analysis only.*

Variables that the generated main must initialize in the cyclic code loop, specified as `'none'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of variable names.

For more information, see `Inputs (-variables-written-in-loop)`.

Example: `opts.CodeProverVerification.VariablesWrittenInLoop = 'all'`

#### CodingRulesCodeMetrics

### `AcAgcSubset` — Subset of MISRA AC AGC rules to check

`'OBL-rules'` (default) | `'OBL-REC-rules'` | `'single-unit-rules'` | `'system-decidable-rules'` | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA AC AGC rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA AC AGC (-misra-ac-agc)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for and Review Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA AC AGC rules, also set `EnableAcAgc` to true.

Example: `opts.CodingRulesCodeMetrics.AcAgcSubset = 'all-rules'`

Data Types: `char`

### `AllowedPragmas` — Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied

cell array of character vectors

Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied, specified as a cell array of character vectors. This property affects only MISRA C:2004 or MISRA AC AGC rule checking.

For more information, see `Allowed pragmas (-allowed-pragmas)`.

Example: `opts.CodingRulesCodeMetrics.AllowedPragmas = {'pragma_01','pragma_02'}`

Data Types: `cell`

### `AutosarCpp14` — Set of AUTOSAR C++ 14 rules to check

`'all'` (default) | `'required'` | `'automated'` | `polyspace.CodingRulesOptions` object | `'from-file'`

*This property affects Bug Finder only.*

Set of AUTOSAR C++ 14 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check AUTOSAR C++ 14 security checks (-autosar-cpp14)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for and Review Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check AUTOSAR C++ 14 rules, also set `EnableAutosarCpp14` to true.

Example: `opts.CodingRulesCodeMetrics.AutosarCpp14 = 'all'`

Data Types: `char`

**`BooleanTypes` — Data types the coding rule checker must treat as effectively Boolean**
cell array of character vectors

Data types that the coding rule checker must treat as effectively Boolean, specified as a cell array of character vectors.

For more information, see `Effective boolean types (-boolean-types)`.

Example: `opts.CodingRulesCodeMetrics.BooleanTypes = {'boolean1_t','boolean2_t'}`

Data Types: `cell`

**`CertC` — Set of CERT C rules and recommendations to check**
`'all'` (default) | `'publish-2016'` | `'all-rules'` | `polyspace.CodingRulesOptions` object | `'from-file'`

*This property affects Bug Finder only.*

Set of CERT C rules and recommendations to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C security checks (-cert-c)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `from-file` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for and Review Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C rules and recommendations, also set `EnableCertC` to true.

Example: `opts.CodingRulesCodeMetrics.CertC = 'all'`

Data Types: `char`

**`CertCpp` — Set of CERT C++ rules to check**
`'all'` (default) | `polyspace.CodingRulesOptions` object | `'from-file'`

*This property affects Bug Finder only.*

Set of CERT C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C++ security checks (-cert-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C++ rules, also set `EnableCertCpp` to true.

Example: `opts.CodingRulesCodeMetrics.CertCpp = 'all'`

Data Types: `char`

### `CheckersSelectionByFile` — File that defines custom set of coding standard checkers

full file path of `.xml` file

File where you define a custom set of coding standards checkers to check, specified as a `.xml` file. You can, in the same file, define a custom set of checkers for each of the coding standards that Polyspace supports. To create a file that defines a custom selection of coding standard checkers, in the Polyspace interface, select a coding standard on the **Coding Standards & Code Metrics** node of the **Configuration** pane and click **Edit**.

For more information, see `Set checkers by file (-checkers-selection-file)`.

Example: `opts.CodingRulesCodeMetrics.CheckersSelectionByFile = 'C:\ps_settings\coding_rules\custom_rules.xml'`

Data Types: `char`

### `CodeMetrics` — Activate code metric calculations

false (default) | true

Activate code metric calculations, specified as true or false. If this property is turned off, Polyspace does not calculate code metrics even if you upload your results to Polyspace Metrics.

For more information about the code metrics, see `Calculate code metrics (-code-metrics)`.

If you assign a coding rules options object to this property, an XML file gets created automatically with the rules specified.

Example: `opts.CodingRulesCodeMetrics.CodeMetrics = true`

### `Cwe` — Set of CWE rules to check

`'all'` (default) | `'cwe-658-659'` | `'from-file'` | `polyspace.CodingRulesOptions` object

*This property affects Bug Finder only.*

Set of CWE rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CWE (-cwe)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C++ rules, also set `EnableCwe` to true.

Example: `opts.CodingRulesCodeMetrics.Cwe = 'cwe-658-659'`

Data Types: `char`

### `EnableAcAgc` — Check MISRA AC AGC rules

false (default) | true

Check MISRA AC AGC rules, specified as true or false. To customize which rules are checked, use `AcAgcSubset`.

For more information about the MISRA AC AGC checker, see `Check MISRA AC AGC (-misra-ac-agc)`.

Example: `opts.CodingRulesCodeMetrics.EnableAcAgc = true;`

### `EnableAutosarCpp14` — Check AUTOSAR C++ 14 rules

false (default) | true

*This property affects Bug Finder only.*

Check AUTOSAR C++ 14 rules, specified as true or false. To customize which rules are checked, use `AutosarCpp14`.

For more information about the AUTOSAR C++ 14 checker, see `Check AUTOSAR C++ 14 checks (-autosar-cpp14)`.

Example: `opts.CodingRulesCodeMetrics.EnableAutosarCpp14 = true;`

### `EnableCertC` — check CERT C rules and recommendations

false (default) | true

*This property affects Bug Finder only.*

Check CERT C rules and recommendations, specified as true or false. To customize which rules are checked, use `CertC`.

For more information about the CERT C checker, see `Check SEI CERT-C checks (-cert-c)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertC = true;`

### `EnableCertCpp` — check CERT C++ rules

false (default) | true

*This property affects Bug Finder only.*

Check CERT C++ rules, specified as true or false. To customize which rules are checked, use `CertCpp`.

For more information about the CERT C++ checker, see `Check SEI-CERT-C++ (-cert-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertCpp = true;`

**`EnableCheckersSelectionByFile` — Check custom set of coding standard checkers**
false (default) | true

Check custom set of coding standard checkers, specified as true or false. If you set this property to true, specify the path of checkers selection XML file for the property `CheckersSelectionByFile` and set one of these properties to 'from-file' to enable the corresponding coding standard custom set:

- `opts.CodingRulesCodeMetrics.AcAgcSubset='from-file'`
- `opts.CodingRulesCodeMetrics.AutosarCpp14='from-file'`
- `opts.CodingRulesCodeMetrics.CertC='from-file'`
- `opts.CodingRulesCodeMetrics.CertCpp='from-file'`
- `opts.CodingRulesCodeMetrics.Cwe='from-file'`
- `opts.CodingRulesCodeMetrics.Guidelines='from-file'`
- `opts.CodingRulesCodeMetrics.Iso17961='from-file'`
- `opts.CodingRulesCodeMetrics.JsfSubset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraC3Subset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCSubset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCppSubset='from-file'`

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;`

**`EnableCustomRules` — Check custom coding rules**
false (default) | true

Check custom coding rules, specified as true or false. The file you specify with `CheckersSelectionByFile` defines the custom coding rules.

Use with `EnableCheckersSelectionByFile`.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCustomRules = true;`

**`EnableCwe` — check CWE rules**
false (default) | true

*This property affects Bug Finder only.*

Check CWE rules, specified as true or false. To customize which rules are checked, use `Cwe`.

For more information about the CWE checker, see `Check CWE (-cwe)`.

Example: `opts.CodingRulesCodeMetrics.Cwe = true;`

**`EnableGuidelines` — Check for violations of coding guidelines**
false (default) | true

*This property affects Bug Finder only.*

Specify whether to check for violations of Guidelines. To customize which rules are checked, use `Guidelines`.

For more information about the Guidelines checker, see `Check guidelines (-guidelines)`.

Example: `opts.CodingRulesCodeMetrics.EnableGuidelines = true;`

### `EnableIso17961` — check ISO-17961 rules

false (default) | true

*This property affects Bug Finder only.*

Check ISO/IEC TS 17961 rules, specified as true or false. To customize which rules are checked, use `Iso17961`.

For more information about the ISO-17961 checker, see `Check ISO-17961 security checks (-iso-17961)`.

Example: `opts.CodingRulesCodeMetrics.EnableIso17961 = true;`

### `EnableJsf` — Check JSF C++ rules

false (default) | true

Check JSF C++ rules, specified as true or false. To customize which rules are checked, use `JsfSubset`.

For more information, see `Check JSF C++ rules (-jsf-coding-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableJsf = true;`

### `EnableMisraC` — Check MISRA C:2004 rules

false (default) | true

Check MISRA C:2004 rules, specified as true or false. To customize which rules are checked, use `MisraCSubset`.

For more information, see `Check MISRA C:2004 (-misra2)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC = true;`

### `EnableMisraC3` — Check MISRA C:2012 rules

false (default) | true

Check MISRA C:2012 rules, specified as true or false. To customize which rules are checked, use `MisraC3Subset`.

For more information about the MISRA C:2012 checker, see `Check MISRA C:2012 (-misra3)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC3 = true;`

### `EnableMisraCpp` — Check MISRA C++:2008 rules

false (default) | true

Check MISRA C++:2008 rules, specified as true or false. To customize which rules are checked, use `MisraCppSubset`.

For more information about the MISRA C++:2008 checker, see `Check MISRA C++ rules (-misra-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraCpp = true;`

### `Guidelines` — Set of Guidelines to check

`'all'` (default) | `'his'` | `'from-file'`

*This property affects Bug Finder only.*

Sets of Guideline rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check guidelines (-guidelines)`.
- An XML file specifying Guidelines checkers and their thresholds. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”.

Example: `opts.CodingRulesCodeMetrics.Guidelines = 'his';`

### `Iso17961` — Set of ISO-17961 rules to check

`'all'` (default) | `'decidable'` | `polyspace.CodingRulesOptions` object | `'from-file'`

*This property affects Bug Finder only.*

Set of ISO/IEC TS 17961 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check ISO-17961 (-iso-17961)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check ISO/IEC TS 17961 rules, also set `EnableIso17961` to true.

Example: `opts.CodingRulesCodeMetrics.Iso17961 = 'all'`

Data Types: `char`

### `JsfSubset` — Subset of JSF C++ rules to check

`'shall-rules'` (default) | `'shall-will-rules'` | `'all-rules'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of JSF C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check JSF C++ rules (-jsf-coding-rules)`.

- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check JSF C++ rules, set `EnableJsf` to true.

Example: `opts.CodingRulesCodeMetrics.JsfSubset = 'all-rules'`

Data Types: `char`

### `Misra3AgcMode` — Use the MISRA C:2012 categories for automatically generated code

false (default) | true

Use the MISRA C:2012 categories for automatically generated code, specified as true or false.

For more information, see `Use generated code requirements (-misra3-agc-mode)`.

Example: `opts.CodingRulesCodeMetrics.Misra3AgcMode = true;`

### `MisraC3Subset` — Subset of MISRA C:2012 rules to check

`'mandatory-required'` (default) | `'mandatory'` | `'single-unit-rules'` | `'system-decidable-rules'` | `'all'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C:2012 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2012 (-misra3)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2012 rules, also set `EnableMisraC3` to true.

Example: `opts.CodingRulesCodeMetrics.MisraC3Subset = 'all'`

Data Types: `char`

### `MisraCSubset` — Subset of MISRA C:2004 rules to check

`'required-rules'` (default) | `'single-unit-rules'` | `'system-decidable-rules'` | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C:2004 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2004 (-misra2)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2004 rules, also set `EnableMisraC` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCSubset = 'all-rules'`

Data Types: `char`

### `MisraCppSubset` — Subset of MISRA C++ rules

`'required-rules'` (default) | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C++:2008 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C++ rules (-misra-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See “Check for and Review Coding Standard Violations”. If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C++ rules, set `EnableMisraCpp` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCppSubset = 'all-rules'`

Data Types: `char`

#### EnvironmentSettings

### Dos — Consider that file paths are in MS-DOS style

true (default) | false

Consider that file paths are in MS-DOS style, specified as true or false.

For more information, see `Code from DOS or Windows file system (-dos)`.

Example: `opts.EnvironmentSettings.Dos = true;`

### `IncludeFolders` — Include folders needed for compilation

cell array of include folder paths

Include folders needed for compilation, specified as a cell array of the include folder paths.

To specify all subfolders of a folder, use folder path followed by `**`, for instance, `'C:\includes\**'`. The notation follows the syntax of the `dir` function. See also “Specify Multiple Source Files”.

For more information, see `-I`.

Example: `opts.EnvironmentSettings.IncludeFolders = {'/includes','/com1/inc'};`

Example: `opts.EnvironmentSettings.IncludeFolders = {'C:\project1\common\includes'};`

Data Types: `cell`

### `Includes` — Files to be `#include`-ed by each C file

cell array of files

Files to be `#include`-ed by each C source file in the analysis, specified by a cell array of files.

For more information, see `Include (-include)`.

Example: `opts.EnvironmentSettings.Includes = {'/inc/inc_file.h','/inc/inc_math.h'}`

### `NoExternC` — Ignore linking errors inside extern blocks

false (default) | true

Ignore linking errors inside extern blocks, specified as true or false.

For more information, see `Ignore link errors (-no-extern-c)`.

Example: `opts.EnvironmentSettings.NoExternC = false;`

### `PostPreProcessingCommand` — Command or script to run on source files after preprocessing

character vector

Command or script to run on source files after preprocessing, specified as a character vector of the command to run.

For more information, see `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Example: Linux — `opts.EnvironmentSettings.PostPreProcessingCommand = [pwd,'/replace_keyword.pl']`

Example: Windows — `opts.EnvironmentSettings.PostPreProcessingCommand = '"C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl"'`

### `StopWithCompileError` — Stop analysis if a file does not compile

false (default) | true

Stop analysis if a file does not compile, specified as true or false.

For more information, see `Stop analysis if a file does not compile (-stop-if-compile-error)`.

Example: `opts.EnvironmentSettings.StopWithCompileError = true;`

**InputsStubbing**

### `DataRangeSpecifications` — Constrain global variables, function inputs, and return values of stubbed functions

file path

Constrain global variables, function inputs, and return values of stubbed functions specified by the path to an XML constraint file. For more information about the constraint file, see “Specify External Constraints for Polyspace Analysis”.

For more information about this option, see `Constraint setup (-data-range-specifications)`.

Example: `opts.InputsStubbing.DataRangeSpecifications = 'C:\project\constraint_file.xml'`

### `DoNotGenerateResultsFor` — Files on which you do not want analysis results

`'include-folders'` (default) | `'all-headers'` | `'custom=`*`folder1[,folder2,...]`*`'`

Files on which you do not want analysis results, specified by `'include-folders'`, `'all-headers'`, or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.GenerateResultsFor`. For more information, see `Do not generate results for (-do-not-generate-results-for)`.

Example: `opts.InputsStubbing.DoNotGenerateResultsFor = 'custom=C:\project\file1.c,C:\project\file2.c'`

### `GenerateResultsFor` — Files on which you want analysis results

`'source-headers'` (default) | `'all-headers'` | `'custom=`*`folder1[,folder2,...]`*`'`

Files on which you want analysis results, specified by `'source-headers'`, `'all-headers'`, or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.DoNotGenerateResultsFor`. For more information, see `Generate results for sources and (-generate-results-for)`.

Example: `opts.InputsStubbing.GenerateResultsFor = 'custom=C:\project\includes_common_1,C:\project\includes_common_2'`

### `FunctionsToStub` — Functions to stub during analysis

cell array of function names

*This property affects Code Prover analysis only.*

Functions to stub during analysis, specified as a cell array of function names.

For more information, see `Functions to stub (-functions-to-stub)`.

Example: `opts.InputsStubbing.FunctionsToStub = {'func1', 'func2'}`

### `NoDefInitGlob` — Consider global variables as uninitialized

false (default) | true

*This property affects Code Prover analysis only.*

Consider global variables as uninitialized, specified as true or false.

For more information, see `Ignore default initialization of global variables (-no-def-init-glob)`.

Example: `opts.InputsStubbing.NoDefInitGlob = true`

### `StubECoderLookupTables` — Specify that the analysis must stub functions in the generated code that use lookup tables

true (default) | false

*This property applies only to a Code Prover analysis of code generated from models.*

Specify that the analysis must stub functions in the generated code that use lookup tables. By replacing the functions with stubs, the analysis assumes more precise return values for the functions.

For more information, see `Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)`.

Example: `opts.InputsStubbing.StubECoderLookupTables = true`

#### Macros

### `DefinedMacros` — Macros to be replaced

cell array of macros

In preprocessed code, macros are replaced by the definition, specified in a cell array of macros and definitions. Specify the macro as `Macro=Value`. If you want Polyspace to ignore the macro, leave the `Value` blank. A macro with no equal sign replaces all instances of that macro by 1.

For more information, see `Preprocessor definitions (-D)`.

Example: `opts.Macros.DefinedMacros = {'uint32=int','name3=','var'}`

### `UndefinedMacros` — Macros to undefine

cell array of macros

In preprocessed code, macros are undefined, specified by a cell array of macros to undefine.

For more information, see `Disabled preprocessor definitions (-U)`.

Example: `opts.Macros.DefinedMacros = {'name1','name2'}`

#### MergedComputingSettings

### `BatchBugFinder` — Send Bug Finder analysis to remote server

false (default) | true

*This property affects Bug Finder analysis only.*

Send Bug Finder analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see:

- `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`
- “Run Analysis on Server”

Example: `opts.MergedComputingSettings.BatchBugFinder = true;`

**`BatchCodeProver` — Send Code Prover analysis to remote server**
false (default) | true

*This property affects Code Prover analysis only.*

Send Code Prover analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see:

- `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`
- “Run Analysis on Server”

Example: `opts.MergedComputingSettings.BatchCodeProver = true;`

**`FastAnalysis` — Run Bug Finder analysis using faster local mode**
false (default) | true

*This property affects Bug Finder analysis only.*

Use fast analysis mode for Bug Finder analysis, specified as true or false.

For more information, see `Use fast analysis mode for Bug Finder (-fast-analysis)`.

Example: `opts.MergedComputingSettings.FastAnalysis = true;`

**MergedReporting**

**`EnableReportGeneration` — Generate a report after the analysis**
false (default) | true

After the analysis, generate a report, specified as true or false.

For more information, see `Generate report`.

Example: `opts.MergedReporting.EnableReportGeneration = true`

**`ReportOutputFormat` — Output format of generated report**
`'Word'` (default) | `'HTML'` | `'PDF'`

Output format of generated report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Output format (-report-output-format)`.

Example: `opts.MergedReporting.ReportOutputFormat = 'PDF'`

**`BugFinderReportTemplate` — Template for generating Bug Finder analysis report**
`'BugFinderSummary'` (default) | `'BugFinder'` | `'SecurityCWE'` | `'CodeMetrics'` | `'CodingStandards'`

*This property affects a Bug Finder analysis only.*

Template for generating analysis report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.BugFinderReportTemplate = 'CodeMetrics'`

### `CodeProverReportTemplate` — Template for generating Code Prover analysis report

`'Developer'` (default) | `'CallHierarchy'` | `'CodeMetrics'` | `'CodingStandards'` | `'DeveloperReview'` | `'Developer_withGreenChecks'` | `'Quality'` | `'VariableAccess'`

*This property affects a Code Prover analysis only.*

Template for generating analysis report, specified as one of the predefined report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.CodeProverReportTemplate = 'CodeMetrics'`

#### Multitasking

### `ArxmlMultitasking` — Specify path of ARXML files to parse for multitasking configuration

cell array of file paths

Specify the path to the ARXML files the software parses to set up your multitasking configuration.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `autosar`.

For more information, see `ARXML files selection (-autosar-multitasking)`

Example: `opts.Multitasking.ArxmlMultitasking={'C:\Polyspace_Workspace\AUTOSAR\myFile.arxml'}`

### `CriticalSectionBegin` — Functions that begin critical sections

cell array of critical section function names

Functions that begin critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionEnd`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionBegin = {'function1:cs1','function2:cs2'}`

### `CriticalSectionEnd` — Functions that end critical sections

cell array of critical section function names

Functions that end critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionBegin`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionEnd = {'function1:cs1','function2:cs2'}`

**`CyclicTasks` — Specify functions that represent cyclic tasks**
cell array of function names

Specify functions that represent cyclic tasks.

To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Cyclic tasks (-cyclic-tasks)`.

Example: `opts.Multitasking.CyclicTasks = {'function1','function2'}`

**`EnableConcurrencyDetection` — Enable automatic detection of certain families of threading functions**
false (default) | true

*This property affects Code Prover analysis only.*

Enable automatic detection of certain families of threading functions, specified as true or false.

For more information, see `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

Example: `opts.Multitasking.EnableConcurrencyDetection = true`

**`EnableExternalMultitasking` — Enable automatic multitasking configuration from external file definitions**
false (default) | true

Enable multitasking configuration of your projects from external files you provide. Configure multitasking from ARXML files for an AUTOSAR project, or from OIL files for an OSEK project.

Activate this option to enable `Multitasking.ArxmlMultitasking` or `Multitasking.OsekMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.EnableExternalMultitasking = 1`

**`EnableMultitasking` — Configure multitasking manually**
false (default) | true

Configure multitasking manually by specifying `true`. This property activates the other manual, multitasking properties.

For more information, see `Configure multitasking manually`.

Example: `opts.Multitasking.EnableMultitasking = 1`

**`EntryPoints` — Functions that serve as entry-points to your multitasking application**
cell array of entry-point function names

Functions that serve as entry-points to your multitasking application specified as a cell array of entry-point function names. To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Tasks (-entry-points)`.

Example: `opts.Multitasking.EntryPoints = {'function1','function2'}`

**`ExternalMultitaskingType` — Specify type of file to parse for multitasking configuration**
`'osek'` (default) | `'autosar'`

Specify the type of file the software parses to set up your multitasking configuration:

- For `osek` type, the analysis looks for OIL files in the file or folder paths that you specify.
- For `autosar` type, the analysis looks for ARXML files in the file paths that you specify.

To activate this option, specify `Multitasking.EnableExternalMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.ExternalMultitaskingType = 'autosar'`

**`Interrupts` — Specify functions that represent nonpreemptable interrupts**
cell array of function names

Specify functions that represent nonpreemptable interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Interrupts (-interrupts)`.

Example: `opts.Multitasking.Interrupts = {'function1','function2'}`

**`InterruptsDisableAll` — Specify routine that disable interrupts**
cell array with one function name

*This property affects Bug Finder analysis only.*

Specify function that disables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsDisableAll = {'function'}`

**`InterruptsEnableAll` — Specify routine that reenable interrupts**
cell array with one function name

*This property affects Bug Finder analysis only.*

Specify function that reenables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsEnableAll = {'function'}`

### `OsekMultitasking` — Specify path of OIL files to parse for multitasking configuration

`'auto'` (default) | `'custom=`*`folder1[,folder2,...]`*`'`

Specify the path to the OIL files the software parses to set up your multitasking configuration:

- In the mode specified with `'auto'`, the analysis uses OIL files in your project source and include folders, but not their subfolders.
- In the mode specified with `'custom=`*`folder1[,folder2,...]`*`'`, the analysis uses the OIL files at the specified path, and the path subfolders.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `osek`.

For more information, see `OIL files selection (-osek-multitasking)`

Example: `opts.Multitasking.OsekMultitasking = 'custom=file_path, dir_path'`

### `TemporalExclusion` — Entry-point functions that cannot execute concurrently

cell array of entry-point function names

Entry-point functions that cannot execute concurrently specified as a cell array of entry-point function names. Each set of exclusive tasks is one cell array entry with functions separated by spaces. To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Temporally exclusive tasks (-temporal-exclusions-file)`.

Example: `opts.Multitasking.TemporalExclusion = {'function1 function2', 'function3 function4 function5'}` where function1 and function2 are temporally exclusive, and function3, function4, and function 5 are temporally exclusive.

#### Precision (Affects Code Prover Only)

### `ContextSensitivity` — Store call context information to identify function call that caused errors

`'none'` (default) | `'auto'` | `'custom=`*`function1[,function2,...]`*`'`

*This property affects Code Prover analysis only.*

Store call context information to identify a function call that caused errors, specified as `none`, `auto`, or as a character array beginning with `custom=` followed by a list of comma-separated function names.

For more information, see `Sensitivity context (-context-sensitivity)`.

Example: `opts.Precision.ContextSensitivity = 'auto'`

Example: `opts.Precision.ContextSensitivity = 'custom=func1'`

### `ModulesPrecision` — Source files you want to verify at higher precision

cell array of file names and precision levels

*This property affects Code Prover analysis only.*

Source files that you want to verify at higher precision, specified as a cell array of file names without the extension and precision levels using this syntax: `filename:Olevel`

For more information, see `Specific precision (-modules-precision)`.

Example: `opts.Precision.ModulesPrecision = {'file1:O0', 'file2:O3'}`

### `OLevel` — Precision level for the verification

2 (default) | 0 | 1 | 3

*This property affects Code Prover analysis only.*

Precision level for the verification, specified as 0, 1, 2, or 3.

For more information, see `Precision level (-O)`.

Example: `opts.Precision.OLevel = 3`

### `PathSensitivityDelta` — Avoid certain verification approximations for code with fewer lines

positive integer

*This property affects Code Prover analysis only.*

Avoid certain verification approximations for code with fewer lines, specified as a positive integer representing how sensitive the analysis is. Higher values can increase verification time exponentially.

For more information, see `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

Example: `opts.Precision.PathSensitivityDelta = 2`

### `Timeout` — Time limit on your verification

character vector

*This property affects Code Prover analysis only.*

Time limit on your verification, specified as a character vector of time in hours.

For more information, see `Verification time limit (-timeout)`.

Example: `opts.Precision.Timeout = '5.75'`

### `To` — Number of times the verification process runs

`'Software Safety Analysis level 2'` (default) | `'Software Safety Analysis level 0'` | `'Software Safety Analysis level 1'` | `'Software Safety Analysis level 3'` | `'Software Safety Analysis level 4'` | `'Source Compliance Checking'` | `'other'`

*This property affects Code Prover analysis only.*

Number of times the verification process runs, specified as one of the preset analysis levels.

For more information, see `Verification level (-to)`.

Example: `opts.Precision.To = 'Software Safety Analysis level 3'`

#### Scaling (Affects Code Prover Only)

### `Inline` — Functions on which separate results must be generated for each function call

cell array of function names

*This property affects Code Prover analysis only.*

Functions on which separate results must be generated for each function call, specified as a cell array of function names.

For more information, see `Inline (-inline)`.

Example: `opts.Scaling.Inline = {'func1','func2'}`

### `KLimiting` — Limit depth of analysis for nested structures

positive integer

*This property affects Code Prover analysis only.*

Limit depth of analysis for nested structures, specified as a positive integer indicating how many levels into a nested structure to verify.

For more information, see `Depth of verification inside structures (-k-limiting)`.

Example: `opts.Scaling.KLimiting = 3`

#### TargetCompiler

### `Compiler` — Compiler that builds your source code

`'generic'` (default) | `'gnu3.4'` | `'gnu4.6'` | `'gnu4.7'` | `'gnu4.8'` | `'gnu4.9'` | `'gnu5.x'` | `'gnu6.x'` | `'gnu7.x'` | `'clang3.x'` | `'clang4.x'` | `'clang5.x'` | `'visual9.0'` | `'visual10'` | `'visual11.0'` | `'visual12.0'` | `'visual14.0'` | `'visual15.x'` | `'keil'` | `'iar'` | `'armcc'` | `'armclang'` | `'codewarrior'` | `'diab'` | `'greenhills'` | `'iar-ew'` | `'renesas'` | `'tasking'` | `'ti'`

Compiler that builds your source code.

For more information, see `Compiler (-compiler)`.

Example: `opts.TargetCompiler.Compiler = 'Visual11.0'`

### `CppVersion` — Specify C++ standard version followed in code

`'defined-by-compiler'` (default) | `'cpp03'` | `'cpp11'` | `'cpp14'` | `'cpp17'`

Specify C++ standard version followed in code, specified as a character vector.

For more information, see `C++ standard version (-cpp-version)`.

Example: `opts.TargetCompiler.CppVersion = 'cpp11';`

### `CVersion` — Specify C standard version followed in code

`'defined-by-compiler'` (default) | `'c90'` | `'c99'` | `'c11'`

Specify C standard version followed in code, specified as a character vector.

For more information, see `C standard version (-c-version)`.

Example: `opts.TargetCompiler.CVersion = 'c90';`

**`DivRoundDown` — Round down quotients from division or modulus of negative numbers**
false (default) | true

Round down quotients from division or modulus of negative numbers, specified as true or false.

For more information, see `Division round down (-div-round-down)`.

Example: `opts.TargetCompiler.DivRoundDown = true`

**`EnumTypeDefinition` — Base type representation of enum**
`'defined-by-compiler'` (default) | `'auto-signed-first'` | `'auto-unsigned-first'`

Base type representation of enum, specified by an allowed base-type set. For more information about the different values, see `Enum type definition (-enum-type-definition)`.

Example: `opts.TargetCompiler.EnumTypeDefinition = 'auto-unsigned-first'`

**`IgnorePragmaPack` — Ignore #pragma pack directives**
false (default) | true

Ignore #pragma pack directives, specified as true or false.

For more information, see `Ignore pragma pack directives (-ignore-pragma-pack)`.

Example: `opts.TargetCompiler.IgnorePragmaPack = true`

**`Language` — Language of analysis**
`'C-CPP'` (default) | `'C'` | `'CPP'`

This property is read-only.

Language of the analysis, specified during the object construction. This value changes which properties appear.

For more information, see `Source code language (-lang)`.

**`LogicalSignedRightShift` — Treatment of signed bit on signed variables**
`'Arithmetical'` (default) | `'Logical'`

Treatment of signed bit on signed variables, specified as `Arithmetical` or `Logical`. For more information, see `Signed right shift (-logical-signed-right-shift)`.

Example: `opts.TargetCompiler.LogicalSignedRightShift = 'Logical'`

**`NoUliterals` — Do not use predefined typedefs for char16_t or char32_t**
false (default) | true

Do not use predefined typedefs for char16_t or char32_t, specified as true or false. For more information, see `Block char16/32_t types (-no-uliterals)`.

Example: `opts.TargetCompiler.NoUliterals = true`

**`PackAlignmentValue` — Default structure packing alignment**
`'defined-by-compiler'` (default) | `'1'` | `'2'` | `'4'` | `'8'` | `'16'`

Default structure packing alignment, specified as `'defined-by-compiler'`, `'1'`, `'2'`, `'4'`, `'8'`, or `'16'`. This property is available only for Visual C++ code.

For more information, see Pack alignment value (-pack-alignment-value).

Example: opts.TargetCompiler.PackAlignmentValue = '4'

### SfrTypes — sfr types

cell array of sfr keywords

sfr types, specified as a cell array of sfr keywords using the syntax *sfr_name=size_in_bits*. For more information, see Sfr type support (-sfr-types).

This option only applies when you set TargetCompiler.Compiler to keil or iar.

Example: opts.TargetCompiler.SfrTypes = {'sfr32=32'}

### SizeTTypeIs — Underlying type of size_t

'defined-by-compiler' (default) | 'unsigned-int' | 'unsigned-long' | 'unsigned-long-long'

Underlying type of size_t, specified as 'defined-by-compiler', 'unsigned-int', 'unsigned-long', or 'unsigned-long-long'. See Management of size_t (-size-t-type-is).

Example: opts.TargetCompiler.SizeTTypeIs = 'unsigned-long'

### Target — Target processor

'i386' (default) | 'arm' | 'arm64' | 'avr' | 'c-167' | 'c166' | 'c18' | 'c28x' | 'c6000' | 'coldfire' | 'hc08' | 'hc12' | 'm68k' | 'mcore' | 'mips' | 'mpc5xx' | 'msp430' | 'necv850' | 'powerpc' | 'powerpc64' | 'rh850' | 'rl78' | 'rx' | 's12z' | 'sharc21x61' | 'sparc' | 'superh' | 'tms320c3x' | 'tricore' | 'x86_64' | generic target object

Set size of data types and endianness of processor, specified as one of the predefined target processors or a generic target object.

For more information about the predefined processors, see Target processor type (-target).

For more information about creating a generic target, see polyspace.GenericTargetOptions.

Example: opts.TargetCompiler.Target = 'hc12'

### WcharTTypeIs — Underlying type of wchar_t

'defined-by-compiler' (default) | 'signed-short' | 'unsigned-short' | 'signed-int' | 'unsigned-int' | 'signed-long' | 'unsigned-long'

Underlying type of wchar_t, specified as 'defined-by-compiler', 'signed-short', 'unsigned-short', 'signed-int', 'unsigned-int', 'signed-long', or 'unsigned-long'. See Management of wchar_t (-wchar-t-type-is).

Example: opts.TargetCompiler.WcharTTypeIs = 'unsigned-int'

#### VerificationAssumption (Affects Code Prover Only)

### ConsiderVolatileQualifierOnFields — Assume that volatile qualified structure fields can have all possible values at any point in code

false (default) | true

*This property affects Code Prover analysis only.*

Assume that volatile qualified structure fields can have all possible values at any point in code.

For more information, see `Consider volatile qualifier on fields (-consider-volatile-qualifier-on-fields)`.

Example: `opts.VerificationAssumption.ConsiderVolatileQualifierOnFields = true`

### `ConstraintPointersMayBeNull` — Specify that environment pointers can be NULL unless constrained otherwise

false (default) | true

*This property affects Code Prover analysis only.*

Specify that environment pointers can be NULL unless constrained otherwise.

For more information, see `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`.

Example: `opts.VerificationAssumption.ConstraintPointersMayBeNull = true`

### `FloatRoundingMode` — Rounding modes to consider when determining the results of floating-point arithmetic

`to-nearest` (default) | `all`

*This property affects Code Prover analysis only.*

Rounding modes to consider when determining the results of floating-point arithmetic, specified as `to-nearest` or `all`.

For more information, see `Float rounding mode (-float-rounding-mode)`.

Example: `opts.VerificationAssumption.FloatRoundingMode = 'all'`

#### Other Properties

### `Author` — Project author

username of current user (default) | character vector

Name of project author, specified as a character vector.

For more information, see `-author`.

Example: `opts.Author = 'JaneDoe'`

### `ImportComments` — Import comments and justifications from previous analysis

character vector

To import comments and justifications from a previous analysis, specify the path to the results folder of the previous analysis.

You can also point to a previous results folder to see only new results compared to the previous run. See “Compare Results from Different Polyspace Runs by Using MATLAB Scripts”.

For more information, see `-import-comments`

Example:
```
opts.ImportComments =
fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example','Module_1','BF_Result')
```

### `Prog` — Project name

`PolyspaceProject` (default) | character vector

Project name, specified as a character vector.

For more information, see `-prog`.

Example: `opts.Prog = 'myProject'`

### `ResultsDir` — Location to store results

folder path

Location to store results, specified as a folder path. By default, the results are stored in the current folder.

For more information, see `-results-dir`.

You can also create a separate results folder for each new run. See “Compare Results from Different Polyspace Runs by Using MATLAB Scripts”.

Example: `opts.ResultsDir = 'C:\project\myproject\results\'`

### `Sources` — Source files

cell array of files

Source files to analyze, specified as a cell array of files.

To specify all files in a folder, use folder path followed by *, for instance, `'C:\src\*'`. To specify all files in a folder and its subfolders, use folder path followed by **, for instance, `'C:\src\**'`. The notation follows the syntax of the `dir` function. See also “Specify Multiple Source Files”.

For more information, see `-sources`.

Example: `opts.Sources = {'file1.c', 'file2.c', 'file3.c'}`

Example: `opts.Sources = {'project/src1/file1.c', 'project/src2/file2.c', 'project/src3/file3.c'}`

### `Version` — Project version number

`'1.0'` (default) | character array of a number

Version number of project, specified as a character array of a number. This option is useful if you upload your results to Polyspace Metrics. If you increment version numbers each time that you reanalyze your object, you can compare the results from two versions in Polyspace Metrics.

For more information, see `-v[ersion]`.

Example: `opts.Version = '2.3'`

## Version History

**Introduced in R2017a**

## See Also

### Topics

“Complete List of Polyspace Bug Finder Analysis Engine Options”

# copyTo

**Class:** `polyspace.Options`
**Package:** `polyspace`

Copy common settings between Polyspace options objects

## Syntax

`optsFrom.copyTo(optsTo)`

## Description

`optsFrom.copyTo(optsTo)` copies the common options from `optsFrom` to `optsTo`. The options objects do not need to be the same type of options object. This method copies only properties that are common between the two objects.

## Input Arguments

**`optsFrom` — Options object you want to copy properties from**
`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object that you want to copy properties from, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

**`optsTo` — Options object you want to copy properties to**
`polyspace.Options` object

Option object that you want to copy properties to, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

## Examples

### Copy Polyspace Options Object

This example shows how to set the properties of one options object and then copy that object to another one.

Create a Polyspace options object and set properties.

```
opts1 = polyspace.Options();
opts1.Prog = 'DataRaceProject';
opts1.Sources = {'datarace.c'};
opts1.TargetCompiler.Compiler = 'gnu4.9';
```

Create another object and use copyTo to copy over options from the previous object.

```
opts2 = polyspace.Options();
opts1.copyTo(opts2);
```

## Version History

**Introduced in R2016b**

### See Also

`polyspace.Options` | `generateProject` | `polyspace.ModelLinkOptions`

# generateProject

**Class:** polyspace.Options
**Package:** polyspace

Generate psprj project from options object

## Syntax

opts.generateProject(projectName)

## Description

opts.generateProject(projectName) creates a .psprj project called projectName from the options specified in the polyspace.Options object opts. You can open a .psprj project in the user interface of the Polyspace desktop products.

## Input Arguments

**opts — Options object to convert into a psprj file**
polyspace.Options or polyspace.ModelLinkOptions object

Option object convert into a psprj file, specified as a polyspace.Options or polyspace.ModelLinkOptions object.

Example: opts = polyspace.Options;

**projectName — Project file name**
character vector

Project file name specified as a character vector. This argument is used as the name of the psprj file.

Example: 'myProject'

## Examples

### Generate Project from a Bug Finder Options Object

This example shows how to create and use a Polyspace project that was generated from an options object.

Create a Bug Finder object and set properties.

```
sources = fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example',...
   'sources','numerical.c');
opts = polyspace.Options();
opts.Prog = 'MyProject';
opts.Sources = {sources};
opts.TargetCompiler.Compiler = 'gnu4.7';
```

Generate a Polyspace project. Name the project using the Prog property.

```
psprj = opts.generateProject(opts.Prog);
```

Run a Bug Finder analysis using one of these commands. Both commands produce identical analysis results. The only difference is that the psprj project can be rerun in the Polyspace interface.

```
polyspaceBugFinder(psprj, '-nodesktop');
polyspaceBugFinder(opts);
```

To run a Code Prover analysis, use polyspaceCodeProver instead of polyspaceBugFinder.

## Tips

If you want to include an options object in a pslinkoptions object:

1. Use this method to convert your object to a project.
2. Add the project to the pslinkoptions property PrjConfig.
3. Turn on the property EnablePrjConfig.

# Version History

**Introduced in R2016b**

## See Also

polyspace.Options | copyTo | polyspace.ModelLinkOptions

# toScript

**Class:** `polyspace.Options`
**Package:** `polyspace`

Add Polyspace options object definition to a script

## Syntax

```
filePath = opts.toScript(fileName,positionInScript)
```

## Description

`filePath = opts.toScript(fileName,positionInScript)` adds the properties of a `polyspace.Options` object to a MATLAB script. The script shows the values assigned to all the properties of the object. You can run the script later to define the object in the MATLAB workspace and use it.

## Input Arguments

### `opts` — Options object with Polyspace analysis options

`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object to store in MATLAB script, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

### `fileName` — Script name

character vector

Name or path to script, specified as a character vector. If you specify a relative path, the script is created in subfolder of the current working folder.

Example: `'runPolyspace.m'`

### `positionInScript` — Where to add object definition

`'create'` (default) | `'append'`

Position in script where the object properties are added, specified as `'create'` or `'append'`. If you specify `'append'`, the object properties are added to the end of an existing script. Otherwise, a new script is created.

## Output Arguments

### `filePath` — Full path to script

character vector

Full path to script, specified as a character vector.

Example: `'C:\myScripts\runPolyspace.m'`

## Version History

**Introduced in R2017b**

### See Also

`polyspace.Options` | `generateProject` | `polyspace.ModelLinkOptions` | `copyTo`

# run

Run a Polyspace analysis

## Syntax

`run(proj, product)`

## Description

`status` = `run(proj, product)` runs a Polyspace Bug Finder or Polyspace Code Prover analysis using the configuration specified in the `polyspace.Project` object `proj`. The analysis results are also stored in `proj`.

## Input Arguments

**`proj` — Polyspace project**
`polyspace.Project` object

Polyspace project with configuration and results, specified as a `polyspace.Project` object.

**`product` — Type of analysis**
`'bugFinder'` | `'codeProver'`

Type of analysis to run.

## Output Arguments

**`status` — Results of a Code Prover analysis**
`true` | `false`

Status of analysis. If the analysis succeeds, the status is `false`. Otherwise, it is `true`.

The analysis can fail for multiple reasons:

- You provide source files that do not exist.
- None of your files compile. Even if one file compiles, unless you set the property `StopWithCompileError` to `true`, the analysis succeeds and returns a `false` status.

There can be many other reasons why the analysis fails. If the analysis fails, in your results folder, check the log file. You can see the results folder using the `Configuration` property of the `polyspace.Project` object:

```
proj = polyspace.Project;
proj.Configuration.ResultsDir
```

The log file is named `Polyspace_R20##`*`n_ProjectName_date-time`*`.log`.

## Examples

### Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
bfSummary = proj.Results.getSummary('defects');
```

# Version History

**Introduced in R2017b**

# getSummary

View number of Polyspace results organized by results type (Bug Finder) or color and file (Code Prover)

## Syntax

```
resSummary = getSummary(resObj, resultsType)
```

## Description

`resSummary = getSummary(resObj, resultsType)` returns the distribution of results of type `resultsType` in a Polyspace results set, `resObj`. The results set `resObj` can be a Bug Finder results set denoted by a `polyspace.BugFinderResults` object or a Code Prover results set denoted by a `polyspace.CodeProverResults` object.

For instance:

- If you choose to see Bug Finder defects, you can see how many defects of each type are present in the result set, for instance, how many non-initialized variables or declaration mismatches.
- If you choose to see Code Prover run-time checks, you see how many red, orange, gray and green checks are present in each file.

## Examples

### Read Existing Bug Finder Results to MATLAB Tables

This example shows how to read Bug Finder analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath=fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example', ...
'Module_1','BF_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.BugFinderResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary(resObj, 'defects');
resTable = getResults(resObj);
```

### Run Bug Finder Analysis and Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
resObj = proj.Results;
bfSummary = getSummary(resObj, 'defects');
```

#### Read Existing Code Prover Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(polyspaceroot,'polyspace','examples','cxx','Code_Prover_Example', ...
'Module_1','CP_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.CodeProverResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary(resObj, 'runtime');
resTable = getResults(resObj);
```

#### Run Code Prover Analysis and Read Results to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a `main` function must be generated, if it does not exist in the source code.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;


% Run analysis
cpStatus = run(proj, 'codeProver');

% Read results
resObj = proj.Results;
cpSummary = getResults(resObj, 'readable');
```

## Input Arguments

### resObj — Bug Finder or Code Prover results

polyspace.BugFinderResults or polyspace.CodeProverResults object

Bug Finder or Code Prover results set, specified as a polyspace.BugFinderResults or polyspace.CodeProverResults object respectively.

### resultsType — Type of Bug Finder or Code Prover analysis result

'defects' | 'runtime' | 'misraC' | 'misraCAGC' | 'misraCPP' | 'misraC2012' | 'jsf' | 'certC' | 'certCpp' | 'iso17961' | 'autosarCPP14' | 'metrics' | 'customRules'

Type of result, specified as a character vector. The default for a Bug Finder results set is 'defects' and the default for a Code Prover results set is 'runtime'.

| Entry | Meaning |
|---|---|
| 'defects' | Bug Finder defects. |
| 'runtime' | Code Prover checks for run-time errors. |
| 'misraC' | MISRA C:2004 rules. |
| 'misraCAGC' | MISRA C:2004 rules for generated code. |
| 'misraCPP' | MISRA C++ rules. |
| 'misraC2012' | MISRA C:2012 rules. |
| 'jsf' | JSF C++ rules. |
| 'certC' | CERT C rules. |
| 'certCpp' | CERT C++ rules. |
| 'iso17961' | ISO/IEC TS 17961 rules. |
| 'autosarCPP14' | AUTOSAR C++ 14 rules. |
| 'metrics' | Code complexity metrics. |
| 'customRules' | Custom rules enforcing naming conventions for identifiers. |

## Output Arguments

### resSummary — Distribution of Bug Finder results by result type or Code Prover run-time checks by check color and file

table

Distribution of results, specified as a table. For instance:

- If you choose to see a summary of Bug Finder defects, an extract of the table looks like this:

  | Category | Defect | Impact | Total |
  |---|---|---|---|
  | Concurrency | Data race | High | 2 |
  | Concurrency | Deadlock | High | 1 |
  | Data flow | Non-initialized variable | High | 2 |

  The table above shows that the result set contains two data races, one deadlock and two non-initialized variables.
- If you choose to see a summary of Code Prover run-time checks, an extract of the table looks like this:

  | File | Proven | Green | Red | Gray | Orange |
  |---|---|---|---|---|---|
  | `file1.c` | 92.0% | 87 | 3 | 2 | 8 |
  | `file2.c` | 97.7% | 41 | 0 | 1 | 1 |

  The table above shows that `file1.c` has:

  - 3 red, 2 gray and 8 orange checks.
  - 92% of operations proven.

    In other words, of every 100 operations that the verification checked, 92 operations were proven green, red or gray. See “Code Prover Result and Source Code Colors” (Polyspace Code Prover).

For more information on MATLAB tables, see “Tables”.

# Version History

**Introduced in R2017a**

## See Also

`polyspace.BugFinderResults` | `polyspace.CodeProverResults`

# getResults

View all instances of Bug Finder or Code Prover results

## Syntax

```
resTable = getResults(resObj, content)
```

## Description

`resTable = getResults(resObj, content)` returns a table showing all results in a Polyspace result set, `resObj`. The results set `resObj` can be a Bug Finder results set denoted by a `polyspace.BugFinderResults` object or a Code Prover results set denoted by a `polyspace.CodeProverResults` object. You can manipulate the table to produce graphs and statistics about your results that you cannot obtain readily from the user interface.

## Examples

### Read Existing Bug Finder Results to MATLAB Tables

This example shows how to read Bug Finder analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example', ...
'Module_1','BF_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.BugFinderResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary(resObj, 'defects');
resTable = getResults(resObj);
```

### Run Bug Finder Analysis and Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
```

```
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = proj.run('bugFinder');

% Read results
resObj = proj.Results;
bfSummary = getResults(resObj, 'readable');
```

### Read Existing Code Prover Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath=fullfile(polyspaceroot,'polyspace','examples','cxx','Code_Prover_Example', ...
'Module_1','CP_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.CodeProverResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary (resObj);
resTable = getResults (resObj);
```

### Run Code Prover Analysis and Read Results to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a `main` function must be generated, if it does not exist in the source code.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;


% Run analysis
cpStatus = proj.run('codeProver');

% Read results
```

```
resObj = proj.Results;
cpSummary = getResults(resObj, 'readable');
```

## Input Arguments

### `resObj` — Bug Finder or Code Prover results

`polyspace.BugFinderResults` or `polyspace.CodeProverResults` object

Bug Finder or Code Prover results set, specified as a `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object respectively.

### `content` — Result information to include

`''` (default) | `'readable'`

Amount of information to be included for each result. If you specify `''`, all information is included. If you specify `'readable'`, the following information is not included:

- ID: Unique number for a result for the current analysis.
- Group: Defect groups, Check groups (Polyspace Code Prover), MISRA C:2012 groups, etc.
- Status, Severity, Comment: Information that *you* enter about a result.

If you do not specify this argument, the full table is included.

See “Export Polyspace Analysis Results”.

## Output Arguments

### `resTable` — Results of a Bug Finder or Code Prover analysis

table

Table showing all results from a single Bug Finder or Code Prover analysis. For each result, the table has information such as file, family, and so on. If a particular information is not available for a result, the entry in the table states `<undefined>`.

For more information on:

- The columns of the table, see “Export Polyspace Analysis Results”.
- MATLAB tables, see “Tables”.

# Version History

**Introduced in R2017a**

## See Also

`polyspace.BugFinderResults` | `polyspace.CodeProverResults`

# Configuration Parameters

# Settings from (C)

Select settings for the analysis configuration. You can quickly activate coding rules checking for generated C code

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** `Project configuration`

`Project configuration`

Run Polyspace with the options specified in the “Project configuration” on page 6-7 or “Use custom project file” on page 6-6.

You do not check coding rules unless you select a rule set in the configuration.

`Project configuration and MISRA AC AGC checking`

Run Polyspace with the options specified in the **Project configuration** plus MISRA AC-AGC obligatory and recommended rules.

`Project configuration and MISRA C 2004 checking`

Run Polyspace with the options specified in the **Project configuration** plus all MISRA C 2004 rules.

`Project configuration and MISRA C 2012 checking`

Run Polyspace with the options specified in the **Project configuration** plus all MISRA C 2012 rules. This option automatically applies the rule categories for generated code. See `Use generated code requirements (-misra3-agc-mode)`.

`MISRA AC AGC checking`

Check compliance with the MISRA AC-AGC obligatory and recommended rules. After rules checking, Polyspace stops.

`MISRA C 2004 checking`

Check compliance with all MISRA C 2004 rules. After rules checking, Polyspace stops.

`MISRA C 2012 checking`

Check compliance with all MISRA C 2012 rules. This option automatically applies the rule categories for generated code. See `Use generated code requirements (-misra3-agc-mode)`. After rules checking, Polyspace stops.

## Dependency

This setting overrides custom configuration settings in “Project configuration” on page 6-7 and “Use custom project file” on page 6-6. If you want to use your custom coding rule settings, select the `Project configuration` option.

## Command-Line Information

Use the `pslinkoptions` property `VerificationSettings`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSVerificationSettings` with the same value as for the `pslinkoptions` property `VerificationSettings`. See pslinkoptions.

## See Also

pslinkoptions | `pslinkoptions`

## Related Examples

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

# Settings from (C++)

Select settings for the analysis configuration. This option allows you to quickly activate coding rules checking for generated C++ code.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** `Project configuration`

`Project configuration`

Run Polyspace with the options specified in the “Project configuration” on page 6-7 or “Use custom project file” on page 6-6.

You do not check coding rules unless you select a rule set in the configuration.

`Project configuration and MISRA C++ checking`

Run Polyspace with the options specified in the **Project configuration** plus MISRA C++ required rules.

`Project configuration and JSF C++ checking`

Run Polyspace with the options specified in the **Project configuration** plus JSF C++ shall rules.

`MISRA C++ checking`

Check compliance with the MISRA C++: 2008 required rules. After rules checking, Polyspace stops.

`JSF C++ checking`

Check compliance with the JSF C++ shall rules. After rules checking, Polyspace stops.

## Dependency

This setting overrides custom configuration settings in “Project configuration” on page 6-7 and “Use custom project file” on page 6-6. If you want to use your custom coding rule settings, select the `Project configuration` option.

## Command-Line Information

Use the `pslinkoptions` property `CxxVerificationSettings`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSCxxVerificationSettings` with the same value as for the `pslinkoptions` property `CxxVerificationSettings`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## Related Examples

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

# Use custom project file

Set Polyspace configuration options with a custom `.psprj` file

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** Off □

Off □

Analysis uses configuration options from **Project configuration** on page 6-7 parameters.

On ☑

Analysis uses configuration options from the specified `.psprj` project file.

## Dependency

The **Settings from** parameter overrides custom configuration settings for coding rules. If you want to use your custom coding rule settings, set **Settings from** > **Project configuration**.

## Command-Line Information

Use the `pslinkoptions` properties `EnablePrjConfigFile` and `PrjConfigFile`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameters `PSEnablePrjConfigFile` and `PSPrjConfigFile` with the same values as for the `pslinkoptions` properties `EnablePrjConfigFile` and `PrjConfigFile`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## Related Examples

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

# Project configuration

Set advanced configuration options to customize the analysis.

## Settings

Open the Polyspace Configuration window by using the **Configure** button. Customize additional settings in this window and save your project configuration. If you added a custom project file in the parameter "Use custom project file" on page 6-6, that project file configuration is shown. Otherwise, the default project template is used.

For details about the advanced options, see "Complete List of Polyspace Bug Finder Analysis Engine Options".

## Dependency

The **Settings from** parameter overrides custom configuration settings for coding rules. If you want to use your custom coding rule settings, set **Settings from > Project configuration**.

## Command-Line Information

Use a Polyspace project (`.psprj` file) with the `pslinkoptions` properties `EnablePrjConfigFile` and `PrjConfigFile`.

## See Also

`polyspace.ModelLinkOptions` | `pslinkoptions` | pslinkoptions

## More About

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

# Enable additional file list

Add additional supporting code files to the analysis.

For instance, suppose you use C files for testing results from the generated code or providing inputs to the generated code. The analysis of generated code only considers files generated from the Simulink model. If you want the analysis to consider the C files that you use for testing or inputs, provide them as additional files.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** Off □

Off □

The analysis includes no additional files.

On ☑

Polyspace analyzes the specified C/C++ files with the generated code. Use the **Select files** button to specify these additional files.

## Command-Line Information

Use the `pslinkoptions` properties `EnableAdditionalFileList` and `AdditionalFileList`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameters `PSEnableAdditionalFileList` and `PSAdditionalFileList` with the same values as for the `pslinkoptions` properties `EnableAdditionalFileList` and `AdditionalFileList`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## More About

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

# Stub lookup tables

Specify that the verification must stub auto-generated functions that use certain kinds of lookup tables in their body. The lookup tables in these functions use linear interpolation and do not allow extrapolation. That is, the result of using the lookup table always lies between the lower and upper bounds of the table.

If you use this option, the verification is more precise and has fewer orange checks. The verification of lookup table functions is usually imprecise. The software has to make certain assumptions about these functions. To avoid missing a run-time error, the verification assumes that the result of using the lookup table is within the full range allowed by the result data type. This assumption can cause many unproven results (orange checks) when a lookup table function is called. By using this option, you narrow down the assumption. For functions using lookup tables with linear interpolation and no extrapolation, the result is at least within the bounds of the table.

The option is relevant only if your model uses Lookup Table blocks.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** On ☑

On ☑

For autogenerated functions that use lookup tables with linear interpolation and no extrapolation, the verification:

- Does not check for run-time errors in the function body.
- Calls a function stub instead of the actual function at the function call sites. The stub ensures that the result of using the lookup table is within the bounds of the table.

To identify if the lookup table in the function uses linear interpolation and no extrapolation, the verification uses information provided by the code generation product. For instance, if you use Embedded Coder to generate code, the lookup table functions with linear interpolation and no extrapolation follow specific naming conventions.

Off □

The verification does not stub autogenerated functions that use lookup tables.

## Tips

- The option applies only to autogenerated functions. If you integrate your own C/C++ S-Function using lookup tables with the model, the option does not cause them to be stubbed.
- The option is on by default. For certification purposes, if you want your verification tool to be independent of the code generation tool, turn off the option.

## Command-Line Information

Use the `pslinkoptions` property `AutoStubLUT`.

The pslinkoptions function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the set_param function to associate this property with the model. Use the parameter PSAutoStubLUT with the same value as for the pslinkoptions property AutoStubLUT. See pslinkoptions.

## See Also

pslinkoptions | pslinkoptions

## More About

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

# Input

Choose whether to constrain Inport block variables.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** `Use specified minimum and maximum values`

`Use specified minimum and maximum values`

Analysis assumes minimum and maximum values for input variables. These values are specified in the Inport block dialog box. Use this value to reduce the number of false positive results.

`Unbounded inputs`

Analysis assumes full range for input variables. Use this value to run a robust analysis that includes values outside the expected range.

## Command-Line Information

Use the `pslinkoptions` property `InputRangeMode`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSInputRangeMode` with the same value as for the `pslinkoptions` property `InputRangeMode`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”
- “External Constraints on Polyspace Analysis of Generated Code”

# Tunable parameters

Choose how to treat tunable parameter values during the analysis. Treat values as either constants or a range of values.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** `Use calibration data`

`Use calibration data`

  Analysis assumes constant values for tunable parameters. Use this value to run a contextual analysis. This option can reduce the number of false positive results.

`Use specified minimum and maximum values`

  Analysis assumes a range of values for the tunable parameter variables. Specify maximum and minimum values in the model. Use this option to run a robust analysis that includes values outside the expected parameter value.

## Command-Line Information

Use the `pslinkoptions` property `ParamRangeMode`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSParamRangeMode` with the same value as for the `pslinkoptions` property `ParamRangeMode`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”
- “External Constraints on Polyspace Analysis of Generated Code”

# Output

Choose whether to verify output values.

Code Prover option only. Bug Finder cannot check output values.

This options is available only for Embedded Coder.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** `No verification`

`No verification`

Polyspace does not verify output values.

`Verify outputs are within minimum and maximum values`

Polyspace checks to see if the output variable values are within the expected minimum and maximum values. Specify the minimum and maximum values in the output block dialog boxes.

## Command-Line Information

Use the `pslinkoptions` property `OutputRangeMode`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSOutputRangeMode` with the same value as for the `pslinkoptions` property `OutputRangeMode`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## More About

- "Run Polyspace Analysis on Code Generated with Embedded Coder"
- "External Constraints on Polyspace Analysis of Generated Code"

# Model reference verification depth

Only for models that use Embedded Coder generated code. Indicate how deep into the model hierarchy to analyze.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** `Current model only`

`Current model only`

Polyspace analyzes only the current model

`1`

Polyspace analyzes the current model and the referenced models that are one level below the current model.

`2`

Polyspace analyzes the current model and the referenced models that are up to two levels below the current model.

`3`

Polyspace analyzes the current model and the referenced models that are up to three levels below the current model.

`All`

Polyspace analyzes the current model and all referenced models.

## Command-Line Information

Use the `pslinkoptions` property `ModelRefVerifDepth`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSModelRefVerifDepth` with the same value as for the `pslinkoptions` property `ModelRefVerifDepth`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

# Model by model verification

Only for models that use Embedded Coder generated code. Analyze each model or referenced model individually. If you have a large project, this option can help modularize your analysis .

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** Off □

Off □

Polyspace analyzes your models together. Model interactions are analyzed.

On ☑

Polyspace analyzes your model and each of its referenced models in isolation. This option does not analyze model interactions.

## Command-Line Information

Use the `pslinkoptions` property `ModelRefByModelRefVerif`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSModelRefByModelRefVerif` with the same value as for the `pslinkoptions` property `ModelRefByModelRefVerif`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

# Output folder

Specify the location and folder name for your analysis results.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** `results_$ModelName$`

Enter a path for your results folder. If you do not use a full path, the results folder is relative to your current MATLAB folder.

If you select “Add results to current Simulink project” on page 6-18, the results folder is relative to the Simulink project folder.

By default, the software stores your results in *`Current Folder`*`\results_`*`model_name`*.

## Command-Line Information

Use the `pslinkoptions` property `ResultDir`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSResultDir` with the same value as for the `pslinkoptions` property `ResultDir`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

# Make output folder name unique by adding a suffix

Add a unique suffix to the results folder for every run to avoid overwriting previous results.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** Off □

Off □

Every time you rerun your analysis, your results are overwritten.

On ☑

For each run of the analysis, Polyspace specifies a new location for the results folder by appending a unique number to the folder name.

## Command-Line Information

Use the `pslinkoptions` property `AddSuffixToResultDir`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSAddSuffixToResultDir` with the same value as for the `pslinkoptions` property `AddSuffixToResultDir`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## More About

- “Run Polyspace Analysis on Code Generated with Embedded Coder”

# Add results to current Simulink project

Add your Polyspace results to the current Simulink project. To use this option, you must have a Simulink project open.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** Off □

Off □

Results are saved to the current folder.

On ☑

Results are saved to the currently open Simulink project.

## Dependencies

You must have a Simulink project open to use this option.

## Command-Line Information

Use the `pslinkoptions` property `AddToSimulinkProject`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSAddToSimulinkProject` with the same value as for the `pslinkoptions` property `AddToSimulinkProject`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## More About

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

# Open results automatically after verification

Decide whether to open your results in the Polyspace interface after running analysis from Simulink.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** On ☑

On ☑

After you run an analysis, your results open automatically in the Polyspace interface.

Off □

You must manually open your results after running an analysis.

## Command-Line Information

Use the `pslinkoptions` property `OpenProjectManager`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSOpenProjectManager` with the same value as for the `pslinkoptions` property `OpenProjectManager`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## More About

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

# Check configuration before verification

Check whether model and code configurations are optimal for code analysis.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** `On (proceed with warnings)`

`On (proceed with warnings)`

The process stops for errors, but continues the code analysis if the configuration has only warnings.

`On (stop for warnings)`

If the configuration has errors or warnings, the process stops.

`Off`

The software does not check the configuration.

## Command-Line Information

Use the `pslinkoptions` property `CheckConfigBeforeAnalysis`. For details, see pslinkoptions.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSVerifALLSFcnInstances` with the same value as for the `pslinkoptions` property `VerifALLSFcnInstances`. See pslinkoptions.

## See Also

`pslinkoptions`

## More About

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

# Verify all S-function occurrences

For S-Function analyses only. Run an analysis on all instances of the selected S-Function.

**Model Configuration Parameters Category**: Polyspace

## Settings

**Default:** Off □

Off □

Analyze only the selected S-Function block. The analysis includes only information from the selected S-Function block.

On ☑

Analyze all occurrences of the S-function in the model. If the S-Function is included in the model multiple times, information from all occurrences is included in the analysis.

## Command-Line Information

Use the `pslinkoptions` property `VerifALLSFcnInstances`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSVerifALLSFcnInstances` with the same value as for the `pslinkoptions` property `VerifALLSFcnInstances`. See pslinkoptions.

## See Also

`pslinkoptions` | pslinkoptions

## More About

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

# Polyspace Results: Defect Checkers

7

# Numerical Defects

# Absorption of float operand

One addition or subtraction operand is absorbed by the other operand

## Description

This defect occurs when one operand of an addition or subtraction operation is *always* negligibly small compared to the other operand. Therefore, the result of the operation is always equal to the value of the larger operand, making the operation redundant.

### Risk

Redundant operations waste execution cycles of your processor.

The absorption of a float operand can indicate design issues elsewhere in the code. It is possible that the developer expected a different range for one of the operands and did not expect the redundancy of the operation. However, the operand range is different from what the developer expects because of issues elsewhere in the code.

### Fix

See if the operand ranges are what you expect. To see the ranges, place your cursor on the operation.

- If the ranges are what you expect, justify why you have the redundant operation in place. For instance, the code is only partially written and you anticipate other values for one or both of the operands from future unwritten code.

  If you cannot justify the redundant operation, remove it.
- If the ranges are not what you expect, in your code, trace back to see where the ranges come from. To begin your traceback, search for instances of the operand in your code. Browse through previous instances of the operand and determine where the unexpected range originates.

To determine when one operand is negligible compared to the other operand, the defect uses rules based on IEEE 754 standards. To fix the defect, instead of using the actual rules, you can use this heuristic: the ratio of the larger to the smaller operand must be less than $2^{p-1}$ at least for some values. Here, `p` is equal to 24 for 32-bit precision and 53 for 64-bit precision. To determine the precision, the defect uses your specification for `Target processor type (-target)`.

This defect appears only if one operand is *always* negligibly smaller than the other operand. To see instances of subnormal operands or results, use the check **Subnormal Float** in Polyspace Code Prover.

## Examples

### One Addition Operand Negligibly Smaller Than The Other Operand

```
#include <stdlib.h>

float get_signal(void);
void do_operation(float);
```

```
float input_signal1(void) {
    float temp = get_signal();
    if(temp > 0. && temp < 1e-30)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

float input_signal2(void) {
    float temp = get_signal();
    if(temp > 1.)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

void main() {
    float signal1 = input_signal1();
    float signal2 = input_signal2();
    float super_signal = signal1 + signal2;
    do_operation(super_signal);
}
```

In this example, the defect appears on the addition because the operand `signal1` is in the range `(0,1e-30)` but `signal2` is greater than `1`.

**Correction — Remove Redundant Operation**

One possible correction is to remove the redundant addition operation. In the following corrected code, the operand `signal2` and its associated code is also removed from consideration.

```
#include <stdlib.h>

float get_signal(void);
void do_operation(float);

float input_signal1(void) {
    float temp = get_signal();
    if(temp > 0. && temp < 1e-30)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

void main() {
    float signal1 = input_signal1();
    do_operation(signal1);
}
```

**Correction — Verify Operand Range**

Another possible correction is to see if the operand ranges are what you expect. For instance, if one of the operand range is not supposed to be negligibly small, fix the issue causing the small range. In the following corrected code, the range `(0,1e-2)` is imposed on `signal2` so that it is not *always* negligibly small as compared to `signal1`.

```
#include <stdlib.h>

float get_signal(void);
void do_operation(float);

float input_signal1(void) {
    float temp = get_signal();
    if(temp > 0. && temp < 1e-2)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

float input_signal2(void) {
    float temp = get_signal();
    if(temp > 1.)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

void main() {
    float signal1 = input_signal1();
    float signal2 = input_signal2();
    float super_signal = signal1 + signal2;
    do_operation(super_signal);
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `FLOAT_ABSORPTION`
**Impact:** High

# Version History

**Introduced in R2016b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Bitwise operation on negative value

Undefined behavior for bitwise operations on negative values

## Description

This defect occurs when bitwise operators (>>, ^, |, ~, but, not, &) are used on signed integer variables with negative values.

### Risk

If the value of the signed integer is negative, bitwise operation results can be unexpected because:

- Bitwise operations on negative values can produce compiler-specific results.
- Unexpected calculations can lead to additional vulnerabilities, such as buffer overflow.

### Fix

When performing bitwise operations, use unsigned integers to avoid unexpected results.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

## Examples

### Right-Shift of Negative Integer

```
#include <stdio.h>
#include <stdarg.h>

static void demo_sprintf(const char *format, ...)
{
    int rc;
    va_list ap;
    char buf[sizeof("256")];

    va_start(ap, format);
    rc = vsprintf(buf, format, ap);
    if (rc == -1 || rc >= sizeof(buf)) {
        /* Handle error */
    }
    va_end(ap);
}

void bug_bitwiseneg()
{
    int stringify = 0x80000000;
    demo_sprintf("%u", stringify >> 24);
}
```

In this example, the statement `demo_sprintf("%u", stringify >> 24)` stops the program unexpectedly. You expect the result of `stringify >> 24` to be `0x80`. However, the actual result is `0xffffff80` because `stringify` is signed and negative. The sign bit is also shifted.

#### Correction — Add `unsigned` Keyword

By adding the `unsigned` keyword, `stringify` is not negative and the right-shift operation gives the expected result of `0x80`.

```
#include <stdio.h>
#include <stdarg.h>

static void demo_sprintf(const char *format, ...)
{
    int rc;
    va_list ap;
    char buf[sizeof("256")];

    va_start(ap, format);
    rc = vsprintf(buf, format, ap);
    if (rc == -1 || rc >= sizeof(buf)) {
        /* Handle error */
    }
    va_end(ap);
}

void corrected_bitwiseneg()
{
    unsigned int stringify = 0x80000000;
    demo_sprintf("%u", stringify >> 24);
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `BITWISE_NEG`
**Impact:** Medium

# Version History

**Introduced in R2016b**

## See Also

`Find defects (-checkers)`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Float conversion overflow

Overflow when converting between floating point data types

## Description

This defect occurs when converting a floating point number to a smaller floating point data type. If the variable does not have enough memory to represent the original number, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

### Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing conversion in subsequent computations and do not account for the overflow, you can see unexpected results.

### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being converted acquires its current value You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller floating point types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and `NaNs`. Operations that results in infinities and `NaNs` might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats).`

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Converting from `double` to `float`

```
float convert(void) {

    double diam = 1e100;
    return (float)diam;
}
```

In the return statement, the variable `diam` of type double (64 bits) is converted to a variable of type float (32 bits). However, the value 1^100 requires more than 32 bits to be precisely represented.

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `FLOAT_CONV_OVFL`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Integer conversion overflow` | `Unsigned integer conversion overflow` | `Sign change integer conversion overflow`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

# Float division by zero

Dividing floating point number by zero

## Description

This defect occurs when the denominator of a division operation can be a zero-valued floating point number.

### Risk

A division by zero can result in a program crash.

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and `NaNs`. Operations that results in infinities and `NaNs` might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats).`

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Dividing a Floating Point Number by Zero

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

#### Correction — Check Before Division

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    if( ((int)denom) != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

#### Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
float fraction(float num)
{
    float denom = 2.0;
    float result = 0.0;

    result = num/denom;

    return result;
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `FLOAT_ZERO_DIV`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Integer division by zero`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Float overflow

Overflow from operation between floating points

## Description

This defect occurs when an operation on floating point variables results in values that cannot be represented by the data type that the operation uses. This data type depends on the operand types and determines the number of bytes allocated for storing the result, thus constraining the range of allowed values.

Note that:

- The data type used to determine an overflow is based on the operand data types. If you then assign the result of an operation to another variable, a different checker, `Float conversion overflow`, determines if the value assigned also overflows the variable assigned to. For instance, in an operation such as:

  ```
  res = x + y;
  ```

  This checker checks for an overflow based on the types of `x` and `y`, and not on the type of `res`. The checker for float conversion overflows then checks for an overflow based on the type of `res`.
- The two operands in a binary operation might undergo promotion before the operation occurs. See also “Code Prover Assumptions About Implicit Data Type Conversions” (Polyspace Code Prover).

The exact storage allocation for different types depends on your processor. See `Target processor type (-target)`.

### Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing computation in subsequent computations and do not account for the overflow, you can see unexpected results.

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and NaNs. Operations that results in infinities and NaNs might be flagged as defects. To handle infinities and NaN values in your code, use the option Consider non finite floats (-allow-non-finite-floats)..

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Multiplication of Floats

```
#include <float.h>

float square(void) {

   float val = FLT_MAX;
   return val * val;
}
```

In the return statement, the variable val is multiplied by itself. The square of the maximum float value cannot be represented by a float (the return type for this function) because the value of val is the maximum float value.

#### Correction — Different Storage Type

One possible correction is to store the result of the operation in a larger data type. In this example, by returning a double instead of a float, the overflow defect is fixed.

```
#include <float.h>

double square(void) {
    float val = FLT_MAX;

    return (double)val * (double)val;
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** FLOAT_OVFL
**Impact:** Low

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Integer overflow` | `Unsigned integer overflow`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

# Integer constant overflow

Constant value falls outside range of integer data type

## Description

This defect occurs in the following cases:

- You assign a compile-time integer constant to a signed integer variable whose data type cannot accommodate the value.

  See "Constant Overflows from Assignments" on page 7-17.
- You use an `enum` value that cannot be accommodated by the underlying type of the `enum` (and the underlying type is signed). For most C compilers, the default underlying type is `signed int` (based on the C standard).

  See "Constant Overflows from enum Values" on page 7-17.
- You perform a binary operation involving two integer constants that results in an overflow, that is, a value outside the range allowed by the data type that the operation uses. A binary operation with integer constants uses the `signed int` data type (unless you use modifiers such as `u` or `L`).

  See "Constant Overflows from Binary Operations" on page 7-18.

An `n`-bit signed integer holds values in the range [-$2^{n-1}$, $2^{n-1}$-1]. For instance, `c` is an 8-bit signed `char` variable that cannot hold the value 255.

```
signed char c = 255;
```

This defect checker depends on the following options:

- `Target processor type (-target)`: Determines the sizes of fundamental types.
- `Enum type definition (-enum-type-definition)`: Determines the underlying types of enumerations.
- `Compiler (-compiler)`: Impacts the interpretation of code.

You do not see the defect in these situations:

- Creation of new constants from `const` variables (for specific compilers only).

  Different compilers might define compile-time constants differently. In the following code, `c+1` is considered a compile-time constant by GCC compilers, but not by the standard C compiler:

  ```
  const int16_t c = 32767;
  int16_t y = c + 1;
  ```

  Whether you see a violation of this check on `y` might depend on your compiler.
- Bitwise `NOT` operation.

  Polyspace does not raise this violation when you perform a bitwise `NOT` operation.

### Risk

The default behavior for constant overflows can vary between compilers and platforms. Retaining constant overflows can reduce the portability of your code.

Even if your compilers wraps around overflowing constants with a warning, the wrap-around behavior can be unintended and cause unexpected results.

### Fix

Check if the constant value is what you intended. If the value is correct, use a different, possibly wider, data type for the variable.

## Examples

### Constant Overflows from Assignments

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    char c1 = MAX_UNSIGNED_CHAR;
    char c2 = MAX_SIGNED_CHAR+1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow. To reproduce these defects, use a `Target processor type (-target)` where `char` is signed by default.

#### Correction — Use Different Data Type

One possible correction is to use a different data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR;
    unsigned char c2 = MAX_SIGNED_CHAR+1;
}
```

### Constant Overflows from `enum` Values

```
enum {
  a=0x7fffffff,
  b,
  c=0xffffffff
} MyEnumA;
```

In this example, the `enum` has an underlying type `int`. The `int` type can accommodate values in the range [-$2^{31}$, $2^{31}$-1]. However, the value of the enumerator `b` is 0x80000000 or $2^{31}$ (one more than the previous value `a`). This value falls outside the allowed range of `int`.

The value of `c`, that is 0xffffffff or $2^{32}$-1, is even larger and also causes an overflow.

To see this defect:

- Specify `Compiler (-compiler)` as `generic`.
- Specify `Source code language (-lang)` as `c`.

**Constant Overflows from Binary Operations**

```
const unsigned int K_ATM_Label_Ram_init_value [] = {
     0x06 | ( 3 << 29) ,
     0x80 | ( 9 << 29) ,
     ( 2 << 31 )        ,
};
```

In this example, two of the shift operations result in values that cannot be accommodated by the `signed int` data type. The `signed int` data type can accommodate values in the range [-$2^{31}$, $2^{31}$-1]. The operation:

- `9 << 29` results in the value $2^{32}$+536870912.
- `2 << 31` results in the value $2^{32}$.

Note that even though the result is assigned to an `unsigned int` variable, the overflow detection uses the underlying type of the binary operation, that is, `signed int`.

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `INT_CONSTANT_OVFL`
**Impact:** Medium

# Version History

**Introduced in R2018b**

## See Also

`Integer overflow` | `Integer conversion overflow` | `Unsigned integer overflow` | `Unsigned integer conversion overflow` | `Unsigned integer constant overflow` | `Sign change integer conversion overflow` | `Find defects (-checkers)`

**Topics**

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Integer conversion overflow

Overflow when converting between integer types

## Description

This defect occurs when converting an integer to a smaller integer type. If the variable does not have enough bytes to represent the original value, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

### Risk

Integer conversion overflows result in undefined behavior.

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Converting from int to char

```
char convert(void) {

    int num = 1000000;

    return (char)num;
}
```

In the return statement, the integer variable num is converted to a char. However, an 8-bit or 16-bit character cannot represent 1000000 because it requires at least 20 bits. So the conversion operation overflows.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number.

```
long convert(void) {

    int num = 1000000;

    return (long)num;
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** INT_CONV_OVFL
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

Float conversion overflow | Unsigned integer conversion overflow | Sign change integer conversion overflow | Find defects (-checkers)

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Integer division by zero

Dividing integer number by zero

## Description

This defect occurs when the denominator of a division or modulo operation can be a zero-valued integer.

### Risk

A division by zero can result in a program crash.

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

#### Correction — Check Before Division

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

#### Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2;
    int result = 0;

    result = num/denom;

    return result;
}
```

### Modulo Operation with Zero

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        arr[i] = input % i;
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

In this example, Polyspace flags the modulo operation as a division by zero. Because modulo is inherently a division operation, the divisor (right hand argument) cannot be zero. The modulo operation uses the `for` loop index as the divisor. However, the `for` loop starts at zero, which cannot be an iterator.

#### Correction — Check Divisor Before Operation

One possible correction is checking the divisor before the modulo operation. In this example, see if the index `i` is zero before the modulo operation.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        if(i != 0)
        {
             arr[i] = input % i;
        }
        else
        {
             arr[i] = input;
        }
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

#### Correction — Change Divisor

Another possible correction is changing the divisor to a nonzero integer. In this example, add one to the index before the `%` operation to avoid dividing by zero.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
         arr[i] = input % (i+1);
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `INT_ZERO_DIV`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Float division by zero`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”
“Extend Bug Finder Checkers to Find Defects from Specific System Input Values”

# Integer overflow

Overflow from operation between integers

## Description

This defect occurs when an operation on integer variables results in values that cannot be represented by the data type that the operation uses. This data type depends on the operand types and determines the number of bytes allocated for storing the result, thus constraining the range of allowed values.

Note that:

- The data type used to determine an overflow is based on the operand data types. If you then assign the result of an operation to another variable, a different checker, `Integer conversion overflow`, determines if the value assigned also overflows the variable assigned to. For instance, in an operation such as:

  ```
  res = x + y;
  ```

  This checker checks for an overflow based on the types of `x` and `y`, and not on the type of `res`. The checker for integer conversion overflows then checks for an overflow based on the type of `res`.
- The two operands in a binary operation might undergo promotion before the operation occurs. See also “Code Prover Assumptions About Implicit Data Type Conversions” (Polyspace Code Prover).

The exact storage allocation for different data types depends on your processor. See `Target processor type (-target)`.

### Risk

Integer overflows on signed integers result in undefined behavior.

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.

- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

## Examples

### Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

#### Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

## Result Information

**Group:** Numerical

**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `INT_OVFL`
**Impact:** Medium

## Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Unsigned integer overflow` | `Float overflow`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Integer precision exceeded

Operation using integer size instead of precision can cause undefined behavior

## Description

This defect occurs when an integer expression uses the integer size in an operation that exceeds the integer precision. On some architectures, the size of an integer in memory can include sign and padding bits. On these architectures, the integer size is larger than the precision which is just the number of bits that represent the value of the integer.

### Risk

Using the size of an integer in an operation on the integer precision can result in integer overflow, wrap around, or unexpected results. For instance, an unsigned integer can be stored in memory in 64 bits, but uses only 48 bits to represent its value. A 56 bits left-shift operation on this integer is undefined behavior.

Assuming that the size of an integer is equal to its precision can also result in program portability issues between different architectures.

### Fix

Do not use the size of an integer instead of its precision. To determine the integer precision, implement a precision computation routine or use a builtin function such as `__builtin_popcount()`.

## Examples

### Using Size of `unsigned int` for Left Shift Operation

```
#include <limits.h>

unsigned int func(unsigned int exp)
{
    if (exp >= sizeof(unsigned int) * CHAR_BIT) {
        /* Handle error */
    }
    return 1U << exp;
}
```

In this example, the function uses a left shift operation to return the value of 2 raised to the power of `exp`. The operation shifts the bits of `1U` by `exp` positions to the left. The `if` statement ensures that the operation does not shift the bits by a number of positions `exp` greater than the size of an `unsigned int`. However, if `unsigned int` contains padding bits, the value returned by `sizeof()` is larger than the precision of `unsigned int`. As a result, some values of `exp` might be too large, and the shift operation might be undefined behavior.

#### Correction — Implement Function to Compute Precision of `unsigned int`

One possible correction is to implement a function `popcount()` that computes the precision of `unsigned int` by counting the number of set bits.

```
#include <stddef.h>
#include <stdint.h>
#include <limits.h>

size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)


unsigned int func(unsigned int exp)
{
    if (exp >= PRECISION(UINT_MAX)) {
        /* Handle error */
    }
    return 1 << exp;
}




size_t popcount(uintmax_t num)
{
    size_t precision = 0;
    while (num != 0) {
        if (num % 2 == 1) {
            precision++;
        }
        num >>= 1;
    }
    return precision;
}
```

### Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `INT_PRECISION_EXCEEDED`
**Impact:** Low

## Version History

**Introduced in R2018b**

### See Also

`Bitwise operation on negative value` | `Possible invalid operation on boolean operand` | `Integer conversion overflow` | `Integer overflow` | `Shift of a negative value` | `Shift operation overflow` | `Unsigned integer conversion overflow` | `Unsigned integer overflow` | `MISRA C:2012 Rule 10.1` | `MISRA C:2012 Rule 10.2` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"

"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Invalid use of standard library floating point routine

Wrong arguments to standard library function

## Description

This defect occurs when you use invalid arguments with a floating point function from the standard library and the standard name space. This defect picks up:

- Rounding and absolute value routines

  `ceil, fabs, floor, fmod`
- Fractions and division routines

  `fmod, modf`
- Exponents and log routines

  `frexp, ldexp, sqrt, pow, exp, log, log10`
- Trigonometry function routines

  `cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh, atanh`

### Risk

Domain errors on standard library floating point functions result in implementation-defined values. If you use the function return value in subsequent computations, you can see unexpected results.

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

It is a good practice to handle for domain errors before using a standard library floating point function. For instance, before calling the `acos` function, check if the argument is in [-1.0, 1.0] and handle the error.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.

- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and `NaNs`. Operations that results in infinities and `NaNs` might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats).`

**Extend Checker**

Extend this checker to check for defects caused by specific values and invalid use of functions from a custom library. For instance:

- You might be using a custom library of mathematical floating point functions. If a custom library function have the same domain and range as another function from the standard library, you can extend this checker to check the custom library function. See “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries”.
- A default Bug Finder analysis might not raise a defect when the input values are unknown and only a subset of inputs can cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Arc Cosine Operation

```
#include <math.h>

double arccosine(void) {
    double degree = 5.0;
    return acos(degree);
}
```

The input value to `acos` must be in the interval `[-1,1]`. This input argument, `degree`, is outside this range.

**Correction — Change Input Argument**

One possible correction is to change the input value to fit the specified range. In this example, change the input value from degrees to radians to fix this defect.

```
#include <math.h>

double arccosine(void) {
    double degree = 5.0;
    double radian = degree * 3.14159 / 180.;
    return acos(radian);
}
```

### Invalid Use of Functions in Standard Namespace

```
#include <cmath>

void InvalidUse(double& val1, double& val2){
  int in = -5;
  int ratio = 0;
  //...
  val1 = std::sqrt(in);
  val2 = std::log(ratio);
}
```

The input value to `std::sqrt()` must be nonnegative. Polyspace flags the use of negative input argument `in`. The input value to `std::log` must be greater than zero. Polyspace flags calling `std::log` by using `ratio`.

#### Correction — Refactor Code

One possible correction is to check the logic and refactor your code. For instance, instead of taking the square root of `-5`, perhaps `val1` is indented to be the negative square root of `5`. The function might be defining `ratio` incorrectly as well.

```
#include <cmath>
int getRatio(void);
void ValidUse(double& val1, double& val2){
  int in = 5;
  int ratio = getRatio();
  //...
  val1 = -1* std::sqrt(in);
  val2 = std::log(ratio);
}
```

### Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `FLOAT_STD_LIB`
**Impact:** High

## Version History

**Introduced in R2013b**

### See Also

`Find defects (-checkers)` | `Invalid use of standard library integer routine` | `Invalid use of standard library memory routine` | `Invalid use of standard library string routine` | `Invalid use of standard library routine`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Invalid use of standard library integer routine

Wrong arguments to standard library function

## Description

This defect occurs when you use invalid arguments with an integer function from the standard library. This defect picks up:

- Character Conversion

  `toupper, tolower`
- Character Checks

  `isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit`
- Integer Division

  `div, ldiv`
- Absolute Values

  `abs, labs`

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

Extend this checker to check for defects caused by specific values and invalid use of functions from a custom library. For instance:

- You might be using a custom library of mathematical functions. If a custom library function have the same domain and range as another function from the standard library, you can extend this checker to check the custom library function. See “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries”.

- A default Bug Finder analysis might not raise a defect when the input values are unknown and only a subset of inputs can cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Absolute Value of Large Negative

```
#include <limits.h>
#include <stdlib.h>

int absoluteValue(void) {

    int neg = INT_MIN;
    return abs(neg);
}
```

The input value to `abs` is `INT_MIN`. The absolute value of `INT_MIN` is `INT_MAX+1`. This number cannot be represented by the type `int`.

#### Correction — Change Input Argument

One possible correction is to change the input value to fit returned data type. In this example, change the input value to `INT_MIN+1`.

```
#include <limits.h>
#include <stdlib.h>

int absoluteValue(void) {

    int neg = INT_MIN+1;
    return abs(neg);
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `INT_STD_LIB`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Invalid use of standard library floating point routine` | `Invalid use of standard library memory routine` | `Invalid use of standard library string routine` | `Invalid use of standard library routine`

**Topics**

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Possible invalid operation on boolean operand

Operation can exceed precision of Boolean operand or result in arbitrary value

## Description

This defect occurs when you use a Boolean operand in an arithmetic, relational, or bitwise operation and:

- The Boolean operand has a trap representation. The size of a Boolean type in memory is at least one addressable unit (size of `char`). A Boolean type requires only one bit to represent the value `true (1)` or `false (0)`. The representation of a Boolean operand in memory contains padding bits. The memory representation can result in values that are not `true` or `false`, a trap representation.
- The result of the operation can exceed the precision of the Boolean operand.

For example, in this code snippet:

```
bool_v >> 2
```

- If the value of `bool_v` is `true (1)` or `false (0)`, the bitwise shift exceeds the one-bit precision of `bool_v` and always results in `0`.
- If `bool_v` has a trap representation, the result of the operation is an arbitrary value.

**Possible invalid operation on boolean operand** raises no defect when:

- The operation does not result in a precision overflow. For instance, bitwise & or `|` operations with `0x01` or `0x00`.
- The Boolean operand cannot have a trap representation. For instance, a constant expression that results in `0` or `1`, or a comparison evaluated to `true` or `false`.

### Risk

Arithmetic, relational, or bitwise operations on a Boolean operand can exceed the operand precision and cause unexpected results when used as a Boolean value. Operations on Boolean operands with trap representations can return arbitrary values.

### Fix

Avoid performing operations on Boolean operands other than these operations:

- Assignment operation (=).
- Equality operations (== or `!=`).
- Logical operations (&&, `||`, or `!`).

## Examples

### Possible Trap Representation of Boolean Operand

```
#include <stdio.h>
#include <stdbool.h>
```

```
#define BOOL _Bool

int arr[2] = {1, 2};

int func(BOOL b)
{
    return arr[b];
}

int main(void)
{
    BOOL b;
    char* ptr = (char*)&b;
    *ptr = 64;
    return func(b);
}
```

In this example, Boolean operand `b` is used as an array index in `func` for an array with two elements. Depending on the compiler and optimization flags you use, the value `b` might not be `0` or `1`. For instance, in Linux Debian 8, if you use `gcc` version 4.9 with optimization flag `-O0`, the value of `b` is 64, which causes a buffer overflow.

**Correction — Use Only Last Significant Bit Value of Boolean Operand**

One possible correction is to use a variable `b0` of type `unsigned int` to get only the value of the last significant bit of the Boolean operand. The value of this bit is in the range `[0..1]`, even if the Boolean operand has a trap representation.

```
#include <stdio.h>
#include <stdbool.h>

#define BOOL _Bool

int arr[2] = {1, 2};

int func(BOOL b)
{
    unsigned int b0 = (unsigned int)b;
    b0 &= 0x1;
    return arr[b0];
}

int main(void)
{
    BOOL b;
    char* ptr = (char*)&b;
    *ptr = 64;
    return func(b);
}
```

Note that a trap representation is often the result of an earlier issue in the code, such as:

- A non-initialized variable of bool type.
- A side effect that modifies any part of a bool type object using a lvalue expression.
- A read of a bool member from a union type with the last stored value of another type.

As such, it is best practice to respect boolean semantics even in C++ code.

### <= Operation Uses Boolean Operands

```
#include <iostream>

template <typename T>
bool less_or_equal(const T& x, const T& y)
{
    std::cout << "INTEGER VERSION" << '\n';
    return x <= y;
}
bool b1 = true, b2 = false;
int i1 = 2, i2 = 3;

int main()
{
    std::cout << std::boolalpha;
    std::cout << "less_or_equal(" << b1 << ',' << b2 << ") = " << less_or_equal<bool>(b1, b2) << '\n';
    std::cout << "less_or_equal(" << i1 << ',' << i2 << ") = " << less_or_equal<int>(11, 12) << '\n';
    return 0;
}
```

In this example, function template `less_or_equal` evaluates whether variable x is less than or equal to y. When you pass boolean types to this function, the <= operation might result in an arbitrary value if the memory representation of the operands, including their padding bits, is neither 1 nor 0.

#### Correction — Specialize Function Template for Boolean Types

One possible correction is to specialize the function template for boolean types. The specialized function template uses a logical (`||`) operation to compare the boolean operands.

```
#include <iostream>

template <typename T>
bool less_or_equal(const T& x, const T& y)
{
    std::cout << "INTEGER VERSION" << '\n';
    return x <= y;
}

template<>
bool less_or_equal<bool>(const bool& x, const bool& y)
{
    std::cout << "BOOLEAN VERSION" << '\n';
    return !x || y;
}

bool b1 = true, b2 = false;
int i1 = 2, i2 = 3;

int main()
{
    std::cout << std::boolalpha;
    std::cout << "less_or_equal(" << b1 << ',' << b2 << ") = " << less_or_equal<bool>(b1, b2) << '\n';
    std::cout << "less_or_equal(" << i1 << ',' << i2 << ") = " << less_or_equal<int>(11, 12) << '\n';
    return 0;
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `INVALID_OPERATION_ON_BOOLEAN`
**Impact:** Low

# Version History

**Introduced in R2018b**

## See Also

Bitwise and arithmetic operation on the same data | Bitwise operation on negative value | Integer conversion overflow | Integer overflow | Integer precision exceeded | Shift of a negative value | Shift operation overflow | Unsigned integer conversion overflow | Unsigned integer overflow | MISRA C:2004 Rule 12.6 | MISRA C:2012 Rule 10.1 | MISRA C:2012 Rule 12.2 | MISRA C++:2008 Rule 4-5-2 | Find defects (-checkers)

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Precision loss in integer to float conversion

Least significant bits of integer lost during conversion to floating-point type

## Description

This defect occurs when you cast an integer value to a floating-point type that cannot represent the original integer value.

For instance, the `long int` value `1234567890L` is too large for a variable of type `float` .

### Risk

If the floating-point type cannot represent the integer value, the behavior is undefined (see C11 standard, 6.3.1.4, paragraph 2). For instance, least significant bits of the variable value can be dropped leading to unexpected results.

### Fix

Convert to a floating-point type that can represent the integer value.

For instance, if the `float` data type cannot represent the integer value, use the `double` data type instead.

When writing a function that converts an integer to floating point type, before the conversion, check if the integer value can be represented in the floating-point type. For instance, `DBL_MANT_DIG * log2(FLT_RADIX)` represents the number of base-2 digits in the type `double`. Before conversion to the type `double`, check if this number is greater than or equal to the precision of the integer that you are converting. To determine the precision of an integer `num`, use this code:

```
size_t precision = 0;
while (num != 0) {
   if (num % 2 == 1) {
     precision++;
   }
   num >>= 1;
}
```

Some implementations provide a builtin function to determine the precision of an integer. For instance, GCC provides the function `__builtin_popcount`.

## Examples

### Conversion of Large Integer to Floating-Point Type

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  float approx = big;
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

In this C code, the `long int` variable `big` is converted to `float`.

#### Correction — Use a Wider Floating-Point Type

One possible correction is to convert to the `double` data type instead of `float`.

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  double approx = big;
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `INT_TO_FLOAT_PRECISION_LOSS`
**Impact:** Low

# Version History

**Introduced in R2018b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Shift of a negative value

Shift operator on negative value

## Description

This defect occurs when a bit-wise shift is used on a variable that can have negative values.

### Risk

Shifts on negative values overwrite the sign bit that identifies a number as negative. The shift operation can result in unexpected values.

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

To fix the defect, check for negative values before the bit-wise shift operation and perform appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Shifting a negative variable

```
int shifting(int val)
{
    int res = -1;
    return res << val;
}
```

In the return statement, the variable `res` is shifted a certain number of bits to the left. However, because `res` is negative, the shift might overwrite the sign bit.

#### Correction — Change the Data Type

One possible correction is to change the data type of the shifted variable to unsigned. This correction eliminates the sign bit, so left shifting does not change the sign of the variable.

```
int shifting(int val)
{
    unsigned int res = -1;
    return res << val;
}
```

### Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SHIFT_NEG`
**Impact:** Low

## Version History

**Introduced in R2013b**

### See Also

`Find defects (-checkers)` | `Shift operation overflow`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Right operand of shift operation outside allowed bounds

Overflow from shifting operation

## Description

This defect occurs when a shift operation can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different data types depends on your processor. See `Target processor type (-target)`.

### Risk

Shift operation overflows can result in undefined behavior.

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface" or "Interpret Bug Finder Results in Polyspace Access Web Interface".

You can fix the defect by:

- Using a bigger data type for the result of the shift operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

## Examples

### Left Shift of Integer

```
int left_shift(void) {

    int foo = 33;
    return 1 << foo;
}
```

In the return statement of this function, bit-wise shift operation is performed shifting 1 `foo` bits to the left. However, an `int` has only 32 bits, so the range of the shift must be between 0 and 31. Therefore, this shift operation causes an overflow.

#### Correction — Different storage type

One possible correction is to store the shift operation result in a larger data type. In this example, by returning a `long long` instead of an `int`, the overflow defect is fixed.

```
long long left_shift(void) {

    int foo = 33;
    return 1LL << foo;
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SHIFT_OVFL`
**Impact:** Low

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Sign change integer conversion overflow

Overflow when converting between signed and unsigned integers

## Description

This defect occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Convert from `unsigned char` to `char`

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

#### Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `SIGN_CHANGE`
**Impact:** Medium

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Float conversion overflow` | `Unsigned integer conversion overflow` | `Integer conversion overflow`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Unsigned integer constant overflow

Constant value falls outside range of unsigned integer data type

## Description

This defect occurs in the following cases:

- You assign a compile-time constant to an unsigned integer variable whose data type cannot accommodate the value.
- You use an enum value that cannot be accommodated by the underlying type of the enum (and the underlying type is unsigned).

An n-bit unsigned integer holds values in the range [0, $2^n$-1]. For instance, c is an 8-bit unsigned char variable that cannot hold the value 256.

```
unsigned char c = 256;
```

This defect checker depends on the following options:

- To determine the sizes of fundamental types, Bug Finder uses your specification for Target processor type (-target).
- To determine the underlying types of enumerations, Bug Finder uses your specification for Enum type definition (-enum-type-definition).

You do not see the defect in these situations:

- Creation of new constants from const variables (for specific compilers only).

  Different compilers might define compile-time constants differently. In the following code, c+1u is considered a compile time-constant by GCC compilers, but not by the standard C compiler:

  ```
  const uint16_t c = 0xffffu;
  uint16_t y = c + 1u;
  ```

  Whether you see a violation of this check on y might depend on your compiler.
- Bitwise NOT operation.

  Polyspace does not raise this violation when you perform a bitwise NOT operation.

### Risk

The C standard states that overflowing unsigned integers must be wrapped around (see, for instance, the C11 standard, section 6.2.5). However, the wrap-around behavior can be unintended and cause unexpected results.

### Fix

Check if the constant value is what you intended. If the value is correct, use a wider data type for the variable.

## Examples

### Overflows from Assignments

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned short c2 = MAX_UNSIGNED_SHORT + 1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow.

#### Correction — Use Wider Data Type

One possible correction is to use a wider data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned short c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned int c2 = MAX_UNSIGNED_SHORT + 1;
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `UINT_CONSTANT_OVFL`
**Impact:** Low

# Version History

**Introduced in R2018b**

## See Also

`Integer overflow` | `Integer conversion overflow` | `Integer constant overflow` | `Unsigned integer overflow` | `Unsigned integer conversion overflow` | `Sign change integer conversion overflow` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unsigned integer conversion overflow

Overflow when converting between unsigned integer types

## Description

This defect occurs when converting an unsigned integer to a smaller unsigned integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

### Risk

Integer conversion overflows result in undefined behavior.

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Converting from int to char

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum;
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned long)unum;
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `UINT_CONV_OVFL`
**Impact:** Low

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Float conversion overflow` | `Integer conversion overflow` | `Sign change integer conversion overflow`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Unsigned integer overflow

Overflow from operation between unsigned integers

## Description

This defect occurs when an operation on unsigned integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

### Risk

The C11 standard states that unsigned integer overflows result in wrap-around behavior. However, a wrap around behavior might not always be desirable. For instance, if the result of a computation is used as an array size and the computation overflows, the array size is much smaller than expected.

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling. In the error handling code, you can override the default wrap-around behavior for overflows and implement saturation behavior, for instance.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Add One to Maximum Unsigned Integer

```
#include <limits.h>

unsigned int plusplus(void) {

    unsigned uvar = UINT_MAX;
    uvar++;
    return uvar;
}
```

In the third statement of this function, the variable `uvar` is increased by 1. However, the value of `uvar` is the maximum unsigned integer value, so 1 plus the maximum integer value cannot be represented by an `unsigned int`. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `uvar` is reduced by modulo `UINT_MAX`. The result is `uvar = 1`.

#### Correction — Different Storage Type

One possible correction is to store the operation result in a larger data type. In this example, by returning an `unsigned long long` instead of an `unsigned int`, the overflow error is fixed.

```
#include <limits.h>

unsigned long long plusplus(void) {

    unsigned long long ullvar = UINT_MAX;
    ullvar++;
    return ullvar;
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `UINT_OVFL`
**Impact:** Low

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Integer overflow` | `Float overflow`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Use of plain char type for numeric value

Plain `char` variable in arithmetic operation without explicit signedness

## Description

This defect occurs when `char` variables without explicit signedness are used in these ways:

- To store non-`char` constants.
- In an arithmetic operation when the `char` is:
  - A negative value.
  - The result of a sign changing overflow.
- As a buffer offset.

`char` variables without a `signed` or `unsigned` qualifier can be signed or unsigned depending on your compiler.

### Risk

Operations on a plain char can result in unexpected numerical values. If the char is used as an offset, the char can cause buffer overflow or underflow.

### Fix

When initializing a char variable, to avoid implementation-defined confusion, explicitly state whether the char is signed or unsigned.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Divide by `char` Variable

```
#include <stdio.h>

void badplaincharuse(void)
{
    char c = 200;
    int i = 1000;
    (void)printf("i/c = %d\n", i/c);
}
```

In this example, the char variable `c` can be signed or unsigned depending on your compiler. Assuming 8-bit, two's complement character types, the result is either `i/c = 5` (unsigned char) or `i/c = -17` (signed char). The correct result is unknown without knowing the signedness of `char`.

#### Correction — Add signed Qualifier

One possible correction is to add a signed qualifier to char. This clarification makes the operation defined.

```
#include <stdio.h>

void badplaincharuse(void)
{
    signed char c = -56;
    int i = 1000;
    (void)printf("i/c = %d\n", i/c);
}
```

## Result Information

**Group:** Numerical
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** BAD_PLAIN_CHAR_USE
**Impact:** Medium

# Version History

**Introduced in R2016b**

## See Also

Find defects (-checkers)

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

8

# Static Memory Defects

# Arithmetic operation with NULL pointer

Arithmetic operation performed on NULL pointer

## Description

This defect occurs when an arithmetic operation involves a pointer whose value is NULL.

### Risk

Performing pointer arithmetic on a null pointer and dereferencing the resulting pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

### Fix

Check a pointer for NULL before arithmetic operations on the pointer.

If the issue occurs despite an earlier check for NULL, look for intermediate events between the check and the subsequent dereference. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

## Examples

### Arithmetic Operation with NULL Pointer Error

```
#include<stdlib.h>

int Check_Next_Value(int *loc, int val)
 {
  int *ptr = loc, found = 0;

  if (ptr==NULL)
   {
     ptr++;
     /* Defect: NULL pointer shifted */

     if (*ptr==val) found=1;
   }

  return(found);
 }
```

When ptr is a NULL pointer, the code enters the if statement body. Therefore, a NULL pointer is shifted in the statement ptr++.

#### Correction — Avoid NULL Pointer Arithmetic

One possible correction is to perform the arithmetic operation when `ptr` is not `NULL`.

```
#include<stdlib.h>

int Check_Next_Value(int *loc, int val)
 {
  int *ptr = loc, found = 0;

  /* Fix: Perform operation when ptr is not NULL */
  if (ptr!=NULL)
   {
     ptr++;

     if (*ptr==val) found=1;
   }

  return(found);
 }
```

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `NULL_PTR_ARITH`
**Impact:** Low

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Null pointer`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Array access out of bounds

Array index outside bounds during array access

## Description

This defect occurs when an array index falls outside the range `[0...array_size-1]` during array access.

### Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

### Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

#### Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `OUT_BOUND_ARRAY`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Pointer access out of bounds`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Buffer overflow from incorrect string format specifier

String format specifier causes buffer argument of standard library functions to overflow

## Description

This defect occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

### Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

### Fix

Use a format specifier that is compatible with the memory buffer size.

## Examples

### Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

**Correction — Use Smaller Precision in Format Specifier**

One possible correction is to read a smaller number of elements into the buffer.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `STR_FORMAT_BUFFER_OVERFLOW`
**Impact:** High

## Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Destination buffer overflow in string manipulation

Function writes to buffer at offset greater than buffer size

## Description

This defect occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

## Examples

### Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

**Correction — Use `snprintf` Instead of `sprintf`**

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `STRLIB_BUFFER_OVERFLOW`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)` | `Destination buffer underflow in string manipulation`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Destination buffer underflow in string manipulation

Function writes to buffer at a negative offset from beginning of buffer

## Description

This defect occurs when certain string manipulation functions write to their destination buffer argument at a negative offset from the beginning of the buffer.

For instance, for the function `sprintf(char* buffer, const char* format)`, you obtain the `buffer` from an operation `buffer = (char*)arr; ... buffer += offset;`. `arr` is an array and `offset` is a negative value.

### Risk

Buffer underflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer underflow also introduces the risk of code injection.

### Fix

If the destination buffer argument results from pointer arithmetic, see if you are decrementing a pointer. Fix the pointer decrement by modifying either the original value before decrement or the decrement value.

## Examples

### Buffer Underflow in `sprintf` Use

```
#include <stdio.h>
#define offset -2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

In this example, `&buffer[offset]` is at a negative offset from the memory allocated to `buffer`.

#### Correction — Change Pointer Decrementer

One possible correction is to change the value of `offset`.

```
#include <stdio.h>
#define offset 2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";
    sprintf(&buffer[offset], fmt_string);
}
```

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `STRLIB_BUFFER_UNDERFLOW`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)` | `Destination buffer overflow in string manipulation`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Invalid use of standard library memory routine

Standard library memory function called with invalid arguments

## Description

This defect occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

### Risk

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Examples

### Invalid Use of Standard Library Memory Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

#### Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `MEM_STD_LIB`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Invalid use of standard library string routine`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Invalid use of standard library string routine

Standard library string function called with invalid arguments

## Description

This defect occurs when a string library function is called with invalid arguments.

### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Examples

### Invalid Use of Standard Library String Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
{
 char *res;
 char gbuffer[5],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);
 /* Error: Size of text is less than gbuffer */

 return(res);
}
```

The string text is larger in size than gbuffer. Therefore, the function strcpy cannot copy text into gbuffer.

**Correction — Use Valid Arguments**

One possible correction is to declare the destination string gbuffer with equal or larger size than the source string text.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** STR_STD_LIB
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

Find defects (-checkers) | Invalid use of standard library memory routine

**Topics**

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Move operation on const object

`std::move` function is called with object declared `const` or `const&`

## Description

This defect occurs when the `std::move` function is called with an object declared `const` or `const&`.

### Risk

For objects declared `const` or `const&`, unlike what you might expect, the copy constructor is called instead of the move constructor.

### Fix

Avoid calling the `std::move` function on `const` objects. If you want to perform a move operation, cast the `const` object to a non-`const` one and then move the non-`const` object.

Note that this issue might also trigger the checker `Const std::move input may cause a more expensive object copy`. If you decide to justify the issue, you can use the same justification for both results.

## Result Information

**Group:** Programming
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `MOVE_CONST_OBJECT`
**Impact:** High

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Dereference of a null pointer

NULL pointer dereferenced

## Description

This defect occurs when you use a pointer with a value of NULL as if it points to a valid memory location. If you dereference the zero address, such as 0x00, Polyspace considers the null address as equivalent to NULL and raises this defect.

### Risk

Dereferencing a null pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

### Fix

Check a pointer for NULL before dereference.

If the issue occurs despite an earlier check for NULL, look for intermediate events between the check and the subsequent dereference. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Null pointer error

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 int* p=NULL;

 *p=arr[0];
 /* Defect: Null pointer dereference */

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

The pointer p is initialized with value of NULL. However, when the value arr[0] is written to *p, p is assumed to point to a valid memory location.

**Correction — Assign Address to Null Pointer Before Dereference**

One possible correction is to initialize p with a valid memory address before dereference.

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 /* Fix: Assign address to null pointer */
 int* p=&arr[0];

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** NULL_PTR
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

Find defects (-checkers) | Arithmetic operation with NULL pointer | Non-initialized pointer

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Bug Finder Checkers to Find Defects from Specific System Input Values"

# Pointer access out of bounds

Pointer dereferenced outside its bounds

## Description

This defect occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Examples

### Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

#### Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `OUT_BOUND_PTR`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Array access out of bounds`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"

“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Pointer or reference to stack variable leaving scope

Pointer to local variable leaves the variable scope

## Description

This defect occurs when a pointer or reference to a local variable leaves the scope of the variable. For instance:

- A function returns a pointer to a local variable.
- A function performs the assignment `globPtr = &locVar`. `globPtr` is a global pointer variable and `locVar` is a local variable.
- A function performs the assignment `*paramPtr = &locVar`. `paramPtr` is a function parameter that is, for instance, an `int**` pointer and `locVar` is a local `int` variable.
- A C++ method performs the assignment `memPtr = &locVar`. `memPtr` is a pointer data member of the class the method belongs to. `locVar` is a variable local to the method.
- (C++11 and later) A function returns a lambda expression object that captures local variables of the function by reference.

The defect also applies to memory allocated using the `alloca` function. The defect does not apply to static, local variables. Polyspace assumes that the local objects within a function definition are in the same scope.

### Risk

Local variables are allocated an address on the stack. Once the scope of a local variable ends, this address is available for reuse. Using this address to access the local variable value outside the variable scope can cause unexpected behavior.

If a pointer to a local variable leaves the scope of the variable, Polyspace Bug Finder highlights the defect. The defect appears even if you do not use the address stored in the pointer. For maintainable code, it is a good practice to not allow the pointer to leave the variable scope. Even if you do not use the address in the pointer now, someone else using your function can use the address, causing undefined behavior.

### Fix

Do not allow a pointer or reference to a local variable to leave the variable scope.

## Examples

### Pointer to Local Variable Returned from Function

```
void func2(int *ptr) {
    *ptr = 0;
}
```

```
int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In this example, `func1` returns a pointer to local variable `ret`.

In `main`, `ptr` points to the address of the local variable. When `ptr` is accessed in `func2`, the access is illegal because the scope of `ret` is limited to `func1`,

#### Pointer to Local Variable Escapes Through Lambda Expression

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [&] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

In this example, the `createAdder` function defines a lambda expression `adder` that captures the local variable `addThis` by reference. The scope of `addThis` is limited to the `createAdder` function. When the object returned by `createAdder` is called, a reference to the variable `addThis` is accessed outside its scope. When accessed in this way, the value of `addThis` is undefined.

##### Correction – Capture Local Variables by Copy in Lambda Expression Instead of Reference

If a function returns a lambda expression object, avoid capturing local variables by reference in the lambda object. Capture the variables by copy instead.

Variables captured by copy have the same lifetime as the lambda object, but variables captured by reference often have a smaller lifetime than the lambda object itself. When the lambda object is used, these variables accessed outside scope have undefined values.

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [=] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `LOCAL_ADDR_ESCAPE`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Subtraction or comparison between pointers to different arrays

Subtraction or comparison between pointers causes undefined behavior

## Description

This defect occurs when you subtract or compare pointers that are null or that point to elements in different arrays. The relational operators for the comparison are >, <, >=, and <=.

### Risk

When you subtract two pointers to elements in the same array, the result is the difference between the subscripts of the two array elements. Similarly, when you compare two pointers to array elements, the result is the positions of the pointers relative to each other. If the pointers are null or point to different arrays, a subtraction or comparison operation is undefined. If you use the subtraction result as a buffer index, it can cause a buffer overflow.

### Fix

Before you subtract or use relational operators to compare pointers to array elements, check that they are non-null and that they point to the same array.

## Examples

### Subtraction Between Pointers to Elements in Different Arrays

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int end;
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation is undefined unless array nums
    is adjacent to variable end in memory. */
    free_elements = &end - next_num_ptr;
    return free_elements;
}
```

In this example, the array `nums` is incrementally filled. Pointer subtraction is then used to determine how many free elements remain. Unless `end` points to a memory location one past the last element of `nums`, the subtraction operation is undefined.

#### Correction — Subtract Pointers to the Same Array

Subtract the pointer to the last element that was filled from the pointer to the last element in the array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation involves pointers to the same array. */
    free_elements = &(nums[SIZE20 - 1]) - next_num_ptr;

    return free_elements + 1;
}
```

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `PTR_TO_DIFF_ARRAY`
**Impact:** High

# Version History

**Introduced in R2017b**

## See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unreliable cast of function pointer

Function pointer cast to another function pointer with different argument or return type

## Description

This defect occurs when a function pointer is cast to another function pointer that has different argument or return type.

### Risk

If you cast a function pointer to another function pointer with different argument or return type and then use the latter function pointer to call a function, the behavior is undefined.

### Fix

Avoid a cast between two function pointers with mismatch in argument or return types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Unreliable cast of function pointer error

```
#include <stdio.h>
#include <math.h>
#define PI 3.142

double Calculate_Sum(int (*fptr)(double))
{
    double  sum = 0.0;
    double  y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;

    fp = sin;
    sum = Calculate_Sum(fp);
    /* Defect: fp implicitly cast to int(*) (double) */

    printf("sum(sin): %f\n", sum);
    return 0;
}
```

The function pointer `fp` is declared as `double (*)(double)`. However in passing it to function `Calculate_Sum`, `fp` is implicitly cast to `int (*)(double)`.

**Correction — Avoid Function Pointer Cast**

One possible correction is to check that the function pointer in the definition of `Calculate_Sum` has the same argument and return type as `fp`. This step makes sure that `fp` is not implicitly cast to a different argument or return type.

```
#include <stdio.h>
#include <math.h>
# define PI 3.142

/*Fix: fptr has same argument and return type everywhere*/
double Calculate_Sum(double (*fptr)(double))
{
    double  sum = 0.0;
    double y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;


    fp = sin;
    sum = Calculate_Sum(fp);
    printf("sum(sin): %f\n", sum);

    return 0;
}
```

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `FUNC_CAST`

**Impact:** Medium

## Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Unreliable cast of pointer`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Unreliable cast of pointer

Pointer implicitly cast to different data type

## Description

This defect occurs when a pointer is implicitly cast to a data type different from its declaration type. Such an implicit casting can take place, for instance, when a pointer to data type `char` is assigned the address of an integer.

This defect applies only if the code language for the project is C.

### Risk

Casting a pointer to data type different from its declaration type can result in issues such as buffer overflow. If the cast is implicit, it can indicate a coding error.

### Fix

Avoid *implicit* cast of a pointer to a data type different from its declaration type.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Unreliable cast of pointer error

```
#include <string.h>

void Copy_Integer_To_String()
{
 int src[]={1,2,3,4,5,6,7,8,9,10};
 char buffer[]="Buffer_Text";
 strcpy(buffer,src);
 /* Defect: Implicit cast of (int*) to (char*) */
}
```

`src` is declared as an `int*` pointer. The `strcpy` statement, while copying to `buffer`, implicitly casts `src` to `char*`.

#### Correction — Avoid Pointer Cast

One possible correction is to declare the pointer `src` with the same data type as `buffer`.

```
#include <string.h>
 void Copy_Integer_To_String()
{
 /* Fix: Declare src with same type as buffer */
 char *src[10]={"1","2","3","4","5","6","7","8","9","10"};
 char *buffer[10];

 for(int i=0;i<10;i++)
   buffer[i]="Buffer_Text";

 for(int i=0;i<10;i++)
   buffer[i]= src[i];
 }
```

### Result Information

**Group:** Static memory
**Language:** C
**Default:** On
**Command-Line Syntax:** PTR_CAST
**Impact:** Medium

## Version History

**Introduced in R2013b**

### See Also

Find defects (-checkers) | Unreliable cast of function pointer

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Use of automatic variable as putenv-family function argument

putenv-family function argument not accessible outside its scope

## Description

This defect occurs when the argument of a `putenv`-family function is a local variable with automatic duration.

### Risk

The function `putenv(char *string)` inserts a pointer to its supplied argument into the environment array, instead of making a copy of the argument. If the argument is an automatic variable, its memory can be overwritten after the function containing the `putenv()` call returns. A subsequent call to `getenv()` from another function returns the address of an out-of-scope variable that cannot be dereferenced legally. This out-of-scope variable can cause environment variables to take on unexpected values, cause the program to stop responding, or allow arbitrary code execution vulnerabilities.

### Fix

Use `setenv()`/`unsetenv()` to set and unset environment variables. Alternatively, use `putenv`-family function arguments with dynamically allocated memory, or, if your application has no reentrancy requirements, arguments with static duration. For example, a single thread execution with no recursion or interrupts does not require reentrancy. It cannot be called (reentered) during its execution.

## Examples

### Automatic Variable as Argument of `putenv()`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    char env[SIZE1024];
    int retval = sprintf(env, "TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
        /* Handle error */
    }
    /* Environment variable TEST is set using putenv().
    The argument passed to putenv is an automatic variable. */
    retval = putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
}
```

In this example, sprintf() stores the character string TEST=var in env. The value of the environment variable TEST is then set to var by using putenv(). Because env is an automatic variable, the value of TEST can change once func() returns.

#### Correction — Use static Variable for Argument of putenv()

Declare env as a static-duration variable. The memory location of env is not overwritten for the duration of the program, even after func() returns.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024
void func(int var)
{
    /* static duration variable */
    static char env[SIZE1024];
    int retval = sprintf(env,"TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
        /* Handle error */
    }

    /* Environment variable TEST is set using putenv() */
    retval=putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
}
```

#### Correction — Use setenv() to Set Environment Variable Value

To set the value of TEST to var, use setenv().

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    /* Environment variable TEST is set using setenv() */
    int retval = setenv("TEST", var ? "1" : "0", 1);

    if (retval != 0) {
        /* Handle error */
    }
}
```

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** On for handwritten code, off for generated code

**Command-Line Syntax:** `PUTENV_AUTO_VAR`
**Impact:** High

## Version History

**Introduced in R2017b**

## See Also

`Pointer or reference to stack variable leaving scope` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Use of path manipulation function without maximum-sized buffer checking

Destination buffer of getwd or realpath is smaller than PATH_MAX bytes

## Description

This defect occurs when the destination argument of a path manipulation function such as realpath or getwd has a buffer size less than PATH_MAX bytes.

### Risk

A buffer smaller than PATH_MAX bytes can overflow but you cannot test the function return value to determine if an overflow occurred. If an overflow occurs, following the function call, the content of the buffer is undefined.

For instance, char *getwd(char *buf) copies an absolute path name of the current folder to its argument. If the length of the absolute path name is greater than PATH_MAX bytes, getwd returns NULL and the content of *buf is undefined. You can test the return value of getwd for NULL to see if the function call succeeded.

However, if the allowed buffer for buf is less than PATH_MAX bytes, a failure can occur for a smaller absolute path name. In this case, getwd does not return NULL even though a failure occurred. Therefore, the allowed buffer for buf must be PATH_MAX bytes long.

### Fix

Possible fixes are:

- Use a buffer size of PATH_MAX bytes. If you obtain the buffer from an unknown source, before using the buffer as argument of getwd or realpath function, make sure that the size is less than PATH_MAX bytes.
- Use a path manipulation function that allows you to specify a buffer size.

  For instance, if you are using getwd to get the absolute path name of the current folder, use char *getcwd(char *buf, size_t size); instead. The additional argument size allows you to specify a size greater than or equal to PATH_MAX.
- Allow the function to allocate additional memory dynamically, if possible.

  For instance, char *realpath(const char *path, char *resolved_path); dynamically allocates memory if resolved_path is NULL. However, you have to deallocate this memory later using the free function.

## Examples

### Possible Buffer Overflow in Use of getwd Function

```
#include <unistd.h>
#include <linux/limits.h>
```

```
#include <stdio.h>

void func(void) {
    char buf[PATH_MAX];
    if (getwd(buf+1)!= NULL)         {
        printf("cwd is %s\n", buf);
    }
}
```

In this example, although the array `buf` has `PATH_MAX` bytes, the argument of `getwd` is `buf + 1`, whose allowed buffer is less than `PATH_MAX` bytes.

#### Correction — Use Array of Size PATH_MAX Bytes

One possible correction is to use an array argument with size equal to `PATH_MAX` bytes.

```
#include <unistd.h>
#include <linux/limits.h>
#include <stdio.h>

void func(void) {
    char buf[PATH_MAX];
    if (getwd(buf)!= NULL)         {
        printf("cwd is %s\n", buf);
    }
}
```

## Result Information

**Group:** Static memory
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `PATH_BUFFER_OVERFLOW`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Wrong allocated object size for cast

Allocated memory does not match destination pointer

## Description

This defect occurs during pointer conversion when the pointer's address is misaligned. If a pointer is converted to a different pointer type, the size of the allocated memory must be a multiple of the size of the destination pointer.

### Risk

Dereferencing a misaligned pointer has undefined behavior and can cause your program to crash.

### Fix

Suppose you convert a pointer `ptr1` to `ptr2`. If `ptr1` points to a buffer of `N` bytes and `ptr2` is a *`type`* `*` pointer where `sizeof(`*`type`*`)` is `n` bytes, make sure that `N` is an integer multiple of `n`.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Dynamic Allocation of Pointers

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(13);
    long *dest;

    dest = (long*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to a `long*`. The dynamically allocated memory of `ptr`, 13 bytes, is not a multiple of the size of `dest`, 4 bytes. This misalignment causes the **Wrong allocated object size for cast** defect.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the allocated memory to 12 instead of 13.

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(12);
    long *dest;

    dest = (long*)ptr;
}
```

### Static Allocation of Pointers

```
void static_non_align(void){
    char arr[13], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to an `int*` in line 6. `ptr` has a memory size of 13 bytes because the array `arr` has a size of 13 bytes. The size of `dest` is 4 bytes, which is not a multiple of 13. This misalignment causes the **Wrong allocated object size for cast** defect.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the size of the array `arr` to a multiple of 4.

```
void static_non_align(void){
    char arr[12], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr;
}
```

### Allocation with a Function

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(13);  //defect
    dest2 = (char*)my_alloc(13); //not a defect
}
```

In this example, the software raises a defect on the conversion of the pointer returned by `my_alloc(13)` to an `int*` in line 11. `my_alloc(13)` returns a pointer with a dynamically allocated size of 13 bytes. The size of `dest1` is 4 bytes, which is not a divisor of 13. This misalignment causes the **Wrong allocated object size for cast** defect. In line 12, the same function call, `my_alloc(13)`, does not call a defect for the conversion to `dest2` because the size of `char*`, 1 byte, a divisor of 13.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the argument for `my_alloc` to a multiple of 4.

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(12);
    dest2 = (char*)my_alloc(13);
}
```

## Result Information

**Group:** Static Memory
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `OBJECT_SIZE_MISMATCH`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Unreliable cast of pointer`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

9

# Dynamic Memory Defects

# Alignment changed after memory reallocation

Memory reallocation changes the originally stricter alignment of an object

## Description

This defect occurs when you use `realloc()` to modify the size of objects with strict memory alignment requirements.

### Risk

The pointer returned by `realloc()` can be suitably assigned to objects with less strict alignment requirements. A misaligned memory allocation can lead to buffer underflow or overflow, an illegally dereferenced pointer, or access to arbitrary memory locations. In processors that support misaligned memory, the allocation impacts the performance of the system.

### Fix

To reallocate memory:

1. Resize the memory block.
   - In Windows, use `_aligned_realloc()` with the alignment argument used in `_aligned_malloc()` to allocate the original memory block.
   - In UNIX/Linux, use the same function with the same alignment argument used to allocate the original memory block.
2. Copy the original content to the new memory block.
3. Free the original memory block.

**Note** This fix has implementation-defined behavior. The implementation might not support the requested memory alignment and can have additional constraints for the size of the new memory.

## Examples

### Memory Reallocated Without Preserving the Original Alignment

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE1024 1024

void func(void)
{
    size_t resize = SIZE1024;
    size_t alignment = 1 << 12; /* 4096 bytes alignment */
    int *ptr = NULL;
    int *ptr1;

    /* Allocate memory with 4096 bytes alignment */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int)) != 0)
    {
        /* Handle error */
      }

    /*Reallocate memory without using the original alignment.
    ptr1 may not be 4096 bytes aligned. */

    ptr1 = (int *)realloc(ptr, sizeof(int) * resize);

    if (ptr1 == NULL)
    {
        /* Handle error */
    }

    /* Processing using ptr1 */

    /* Free before exit */
    free(ptr1);
}
```

In this example, the allocated memory is 4096-bytes aligned. `realloc()` then resizes the allocated memory. The new pointer `ptr1` might not be 4096-bytes aligned.

**Correction — Specify the Alignment for the Reallocated Memory**

When you reallocate the memory, use `posix_memalign()` and pass the alignment argument that you used to allocate the original memory.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE1024 1024

void func(void)
{
    size_t resize = SIZE1024;
    size_t alignment = 1 << 12; /* 4096 bytes alignment */
    int *ptr = NULL;

    /* Allocate memory with 4096 bytes alignment */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int)) != 0)
    {
        /* Handle error */
    }

    /* Reallocate memory using the original alignment. */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int) * resize) != 0)
    {
        /* Handle error */
        free(ptr);
        ptr = NULL;
    }
    /* Processing using ptr */

    /* Free before exit */
    free(ptr);
}
```

## Result Information

**Group:** Dynamic memory
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `ALIGNMENT_CHANGE`
**Impact:** Low

# Version History

**Introduced in R2017b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Deallocation of previously deallocated pointer

Memory freed more than once without allocation

## Description

This defect occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

### Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to free this block of memory can result in a segmentation fault.

### Fix

The fix depends on the root cause of the defect. See if you intended to allocate a memory block to the pointer between the first deallocation and the second. Otherwise, remove the second `free` statement.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before freeing pointers, check them for NULL values and handle the error. In this way, you are protected against freeing an already freed block.

## Examples

### Deallocation of Previously Deallocated Pointer Error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

#### Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>
```

```
void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
 }
```

#### Freeing Pointer Previously Reallocated With Possibly Zero Size

```
#include <stdlib.h>

void reshape(char *buf, size_t size) {
  char *reshaped_buf = (char *)realloc(buf, size);
  if (reshaped_buf == NULL) {
    free(buf);
  }
}
```

In this example, the argument `size` of the `reshape()` function can be zero and result in a zero-size reallocation with `realloc()`. In some implementations such as the GNU library, zero-size reallocations free the memory leading to a double free defect.

##### Correction – Guard Against Zero-Size Reallocations

One possible correction is to check size argument of `realloc()` for zero values before use. If the size argument is zero, you can simply free the memory instead of reallocating it.

```
#include <stdlib.h>

void reshape(char *buf, size_t size) {
  if (size != 0) {
    char *reshaped_buf = (char *)realloc(buf, size);
    if (reshaped_buf == NULL) {
      free(buf);
    }
  }
  else {
    free(buf);
  }

}
```

## Result Information

**Group:** Dynamic memory
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `DOUBLE_DEALLOCATION`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Use of previously freed pointer`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Invalid deletion of pointer

Pointer deallocation using `delete` without corresponding allocation using `new`

## Description

This defect occurs when:

- You release a block of memory with the `delete` operator but the memory was previously not allocated with the `new` operator.
- You release a block of memory with the `delete` operator using the single-object notation but the memory was previously allocated as an array with the `new` operator.

This defect applies only to C++ source files.

### Risk

The risk depends on the cause of the issue:

- The `delete` operator releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.
- If you use the single-object notation for `delete` on a pointer that is previously allocated with the array notation for `new`, the behavior is undefined.

The issue can also highlight other coding errors. For instance, you perhaps wanted to use the `delete` operator or a previous `new` operator on a different pointer.

### Fix

The fix depends on the cause of the issue:

- In most cases, you can fix the issue by removing the `delete` statement. If the pointer is not allocated memory from the heap with the `new` operator, you do not need to release the pointer with `delete`. You can simply reuse the pointer as required or let the object be destroyed at the end of its scope.
- In case of mismatched notation for `new` and `delete`, correct the mismatch. For instance, to allocate and deallocate a single object, use this notation:

  ```
  classType* ptr = new classType;
  delete ptr;
  ```

  To allocate and deallocate an array objects, use this notation:

  ```
  classType* p2 = new classType[10];
  delete[] p2;
  ```

If the issue highlights a coding error such as use of `delete` or `new` on the wrong pointer, correct the error.

## Examples

### Deleting Static Memory

```
void assign_ones(void)
{
    int ptr[10];

    for(int i=0;i<10;i++)
        *(ptr+i)=1;

    delete[] ptr;
}
```

The pointer `ptr` is released using the `delete` operator. However, `ptr` points to a memory location that was not dynamically allocated.

#### Correction: Remove Pointer Deallocation

If the number of elements of the array `ptr` is known at compile time, one possible correction is to remove the deallocation of the pointer `ptr`.

```
void assign_ones(void)
{
    int ptr[10];

    for(int i=0;i<10;i++)
        *(ptr+i)=1;
}
```

#### Correction — Add Pointer Allocation

If the number of array elements is not known at compile time, one possible correction is to dynamically allocate memory to the array `ptr` using the `new` operator.

```
void assign_ones(int num)
{
    int *ptr = new int[num];

    for(int i=0; i < num; i++)
        *(ptr+i) = 1;

    delete[] ptr;
   }
```

### Mismatched new and delete

```
int main (void)
{
    int *p_scale = new int[5];

    //more code using scal

    delete p_scale;
}
```

In this example, `p_scale` is initialized to an array of size 5 using `new int[5]`. However, `p_scale` is deleted with `delete` instead of `delete[]`. The `new`-`delete` pair does not match. Do not use `delete` without the brackets when deleting arrays.

### Correction — Match delete to new

One possible correction is to add brackets so the `delete` matches the `new []` declaration.

```
int main (void)
{
    int *p_scale = new int[5];

    //more code using p_scale

    delete[] p_scale;
}
```

### Correction — Match new to delete

Another possible correction is to change the declaration of `p_scale`. If you meant to initialize `p_scale` as 5 itself instead of an array of size 5, you must use different syntax. For this correction, change the square brackets in the initialization to parentheses. Leave the `delete` statement as it is.

```
int main (void)
{
    int *p_scale = new int(5);

    //more code using p_scale

    delete p_scale;
}
```

## Result Information

**Group:** Dynamic memory
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `BAD_DELETE`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Invalid free of pointer` | `Memory leak`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Invalid free of pointer

Pointer deallocation without a corresponding dynamic allocation

## Description

This defect occurs when a block of memory released using the free function was not previously allocated using malloc, calloc, or realloc.

### Risk

The free function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the free function or a previous malloc function on a different pointer.

### Fix

In most cases, you can fix the issue by removing the free statement. If the pointer is not allocated memory from the heap with malloc or calloc, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of free or malloc on the wrong pointer, correct the error.

If the issue occurs because you use the free function to free memory allocated with the new operator, replace the free function with the delete operator.

## Examples

### Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer p is deallocated using the free function. However, p points to a memory location that was not dynamically allocated.

#### Correction — Remove Pointer Deallocation

If the number of elements of the array p is known at compile time, one possible correction is to remove the deallocation of the pointer p.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

**Correction — Introduce Pointer Allocation**

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p`.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

## Result Information

**Group:** Dynamic Memory
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `BAD_FREE`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Invalid deletion of pointer`

**Topics**

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Memory leak

Memory allocated dynamically not freed

## Description

This defect occurs when you do not free a block of memory allocated through `malloc`, `calloc`, `realloc`, or `new`. If the memory is allocated in a function, the defect does not occur if:

- Within the function, you free the memory using `free` or `delete`.
- The function returns the pointer assigned by `malloc`, `calloc`, `realloc`, or `new`.
- The function stores the pointer in a global variable or in a parameter.

### Risk

Dynamic memory allocation functions such as `malloc` allocate memory on the heap. If you do not release the memory after use, you reduce the amount of memory available for another allocation. On embedded systems with limited memory, you might end up exhausting available heap memory even during program execution.

### Fix

Determine the scope where the dynamically allocated memory is accessed. Free the memory block at the end of this scope.

To free a block of memory, use the `free` function on the pointer that was used during memory allocation. For instance:

```
ptr = (int*)malloc(sizeof(int));
...
free(ptr);
```

It is a good practice to allocate and free memory in the same module at the same level of abstraction. For instance, in this example, `func` allocates and frees memory at the same level but `func2` does not.

```
void func() {
  ptr = (int*)malloc(sizeof(int));
  {
    ...
  }
  free(ptr);
}

void func2() {
  {
   ptr = (int*)malloc(sizeof(int));
   ...
  }
  free(ptr);
}
```

See CERT-C Rule MEM00-C.

## Examples

### Dynamic Memory Not Released Before End of Function

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
}
```

In this example, `pi` is dynamically allocated by `malloc`. The function `assign_memory` does not free the memory, nor does it return `pi`.

#### Correction — Free Memory

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
    free(pi);
}
```

#### Correction — Return Pointer from Dynamic Allocation

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

```
#include<stdlib.h>
#include<stdio.h>

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
```

```
        {
            printf("Memory allocation failed");
            return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

#### Memory Leak with New/Delete

```
#define NULL '\0'

void initialize_arr1(void)
{
    int *p_scalar = new int(5);
}

void initialize_arr2(void)
{
    int *p_array = new int[5];
}
```

In this example, the functions create two variables, `p_scalar` and `p_array`, using the `new` keyword. However, the functions end without cleaning up the memory for these pointers. Because the functions used `new` to create these variables, you must clean up their memory by calling `delete` at the end of each function.

##### Correction — Add Delete

To correct this error, add a `delete` statement for every `new` initialization. If you used brackets `[]` to instantiate a variable, you must call delete with brackets as well.

```
#define NULL '\0'

void initialize_arrs(void)
{
    int *p_scalar = new int(5);
    int *p_array = new int[5];

    delete p_scalar;
    p_scalar = NULL;

    delete[] p_array;
    p_scalar = NULL;
}
```

## Result Information

**Group:** Dynamic memory
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `MEM_LEAK`
**Impact:** Medium

## Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Mismatched alloc/dealloc functions on Windows

Improper deallocation function causes memory corruption issues

## Description

This defect occurs when you use a Windows deallocation function that is not properly paired to its corresponding allocation function.

### Risk

Deallocating memory with a function that does not match the allocation function can cause memory corruption or undefined behavior. If you are using an older version of Windows, the improper function can also cause compatibility issues with newer versions.

### Fix

Properly pair your allocation and deallocation functions according to the functions listed in this table.

| Allocation Function | Deallocation Function |
|---|---|
| malloc() | free() |
| realloc() | free() |
| calloc() | free() |
| _aligned_malloc() | _aligned_free() |
| _aligned_offset_malloc() | _aligned_free() |
| _aligned_realloc() | _aligned_free() |
| _aligned_offset_realloc() | _aligned_free() |
| _aligned_recalloc() | _aligned_free() |
| _aligned_offset_recalloc() | _aligned_free() |
| _malloca() | _freea() |
| LocalAlloc() | LocalFree() |
| LocalReAlloc() | LocalFree() |
| GlobalAlloc() | GlobalFree() |
| GlobalReAlloc() | GlobalFree() |
| VirtualAlloc() | VirtualFree() |
| VirtualAllocEx() | VirtualFreeEx() |
| VirtualAllocExNuma() | VirtualFreeEx() |
| HeapAlloc() | HeapFree() |
| HeapReAlloc() | HeapFree() |

## Examples

### Memory Deallocated with Incorrect Function

```
#ifdef _WIN32_
#include <windows.h>
#else
#define _WIN32_
typedef void *HANDLE;
typedef HANDLE HGLOBAL;
typedef HANDLE HLOCAL;
typedef unsigned int UINT;
extern HLOCAL LocalAlloc(UINT uFlags, UINT uBytes);
extern HLOCAL LocalFree(HLOCAL hMem);
extern HGLOBAL GlobalFree(HGLOBAL hMem);
#endif

#define SIZE9 9


void func(void)
{
    /* Memory allocation */
    HLOCAL p = LocalAlloc(0x0000, SIZE9);

    if (p) {
        /* Memory deallocation. */
        GlobalFree(p);

    }
}
```

In this example, memory is allocated with `LocallAlloc()`. The program then erroneously uses `GlobalFree()` to deallocate the memory.

#### Correction — Properly Pair Windows Allocation and Deallocation Functions

When you allocate memory with `LocalAllocate()`, use `LocalFree()` to deallocate the memory.

```
#ifdef _WIN32_
#include <windows.h>
#else
#define _WIN32_
typedef void *HANDLE;
typedef HANDLE HGLOBAL;
typedef HANDLE HLOCAL;
typedef unsigned int UINT;
extern HLOCAL LocalAlloc(UINT uFlags, UINT uBytes);
extern HLOCAL LocalFree(HLOCAL hMem);
extern HGLOBAL GlobalFree(HGLOBAL hMem);
#endif

#define SIZE9 9
void func(void)
{
    /* Memory allocation */
    HLOCAL p = LocalAlloc(0x0000, SIZE9);
    if (p) {
        /* Memory deallocation. */
        LocalFree(p);
    }
}
```

### Result Information

**Group:** Dynamic memory
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `WIN_MISMATCH_DEALLOC`
**Impact:** Low

## Version History

**Introduced in R2017b**

### See Also

`Invalid deletion of pointer` | `Invalid free of pointer` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unprotected dynamic memory allocation

Pointer returned from dynamic allocation not checked for NULL or nullptr value

## Description

This defect occurs when you access dynamically allocated memory without first checking if the prior memory allocation succeeded.

### Risk

When memory is dynamically allocated using malloc, calloc, or realloc, it returns a value NULL if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this NULL value, this access is not protected from failures.

### Fix

Check the return value of malloc, calloc, or realloc for NULL before accessing the allocated memory location.

```
int *ptr = malloc(size * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

## Examples

### Unprotected dynamic memory allocation error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));

  *p = 2;
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function calloc returns NULL to p. Before accessing the memory through p, the code does not check whether p is NULL

#### Correction — Check for NULL Value

One possible correction is to check whether p has value NULL before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
```

```
 {
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

**Unprotected dynamic memory allocation error only on dereference**

```
#include <stdlib.h>
#include<string.h>
typedef struct recordType {
    const char* id;
    const char* data;
} RECORD;

RECORD* MakerecordType(const char *id,unsigned int size){
    RECORD *rec = (RECORD *)calloc(1, sizeof(RECORD));
    rec->id = strdup(id);

    const char *newData = (char *)calloc(1, size);
    rec->data = newData;
    return rec;
}
```

In this example, the checker raises a defect when you dereference the pointer `rec` without checking for a `NULL` value from the prior dynamic memory allocation.

A similar issue happens with the pointer `newData`. However, a defect is not raised because the pointer is not dereferenced but simply copied over to `rec->data`. Simply copying over a possibly null pointer is not an issue by itself. For instance, callers of the `recordType_new` function might check for `NULL` value of `rec->data` before dereferencing, thereby avoiding a null pointer dereference.

## Result Information

**Group:** Dynamic memory
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `UNPROTECTED_MEMORY_ALLOCATION`
**Impact:** Low

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”

“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Use of previously freed pointer

Memory accessed after deallocation

## Description

This defect occurs when you access a block of memory after freeing the block using the `free` function.

### Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

### Fix

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

## Examples

### Use of Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift;
    /* Defect: Reading a freed pointer */

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

#### Correction — Free Pointer After Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

## Result Information

**Group:** Dynamic memory
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `FREED_PTR`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Deallocation of previously deallocated pointer`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

10

# C++ Exception Defects

# Exception caught by value

catch statement accepts an object by value

## Description

This defect occurs when a catch statement accepts an object by value.

### Risk

If a throw statement passes an object and the corresponding catch statement accepts the exception by value, the object is copied to the catch statement parameter. This copy can lead to unexpected behavior such as:

- Object slicing, if the throw statement passes a derived class object.
- Undefined behavior of the exception, if the copy fails.

### Fix

Catch the exception by reference or by pointer. Catching an exception by reference is recommended.

## Examples

### Standard Exception Caught by Value

```
#include <exception>

extern void print_str(const char* p);
extern void throw_exception();

void func() {
    try {
        throw_exception();
    }

    catch(std::exception exc) {
        print_str(exc.what());
    }
}
```

In this example, the catch statement takes a std::exception object by value. Catching an exception by value causes copying of the object. It can cause undefined behavior of the exception if the copy fails.

#### Correction: Catch Exception by Reference

One possible solution is to catch the exception by reference.

```
#include <exception>

extern void print_str(const char* p);
extern void throw_exception();
```

```
void corrected_excpcaughtbyvalue() {
    try {
        throw_exception();
    }
    catch(std::exception& exc) {
        print_str(exc.what());
    }
}
```

### Derived Class Exception Caught by Value

```
#include <exception>
#include <string>
#include <typeinfo>
#include <iostream>

// Class declarations
class BaseExc {
public:
    explicit BaseExc();
    virtual ~BaseExc() {};
protected:
    BaseExc(const std::string& type);
private:
    std::string _id;
};

class IOExc: public BaseExc {
public:
    explicit IOExc();
};

//Class method declarations
BaseExc::BaseExc():_id(typeid(this).name()) {
}
BaseExc::BaseExc(const std::string& type): _id(type) {
}
IOExc::IOExc(): BaseExc(typeid(this).name()) {
}

int input(void);

int main(void) {
    int rnd = input();
    try {
        if (rnd==0) {
            throw IOExc();
        } else {
            throw BaseExc();
        }
    }


    catch(BaseExc exc) {
        std::cout << "Intercept BaseExc" << std::endl;
    }
    return 0;
}
```

In this example, the `catch` statement takes a `BaseExc` object by value. Catching exceptions by value causes copying of the object. The copying can cause:

- Undefined behavior of the exception if it fails.
- Object slicing if an exception of the derived class `IOExc` is caught.

**Correction — Catch Exceptions by Reference**

One possible correction is to catch exceptions by reference.

```
#include <exception>
#include <string>
#include <typeinfo>
#include <iostream>

// Class declarations
class BaseExc {
public:
    explicit BaseExc();
    virtual ~BaseExc() {};
protected:
    BaseExc(const std::string& type);
private:
    std::string _id;
};

class IOExc: public BaseExc {
public:
    explicit IOExc();
};

//Class method declarations
BaseExc::BaseExc():_id(typeid(this).name()) {
}
BaseExc::BaseExc(const std::string& type): _id(type) {
}
IOExc::IOExc(): BaseExc(typeid(this).name()) {
}

int input(void);

int main(void) {
    int rnd = input();
    try {
        if (rnd==0) {
            throw IOExc();
        } else {
            throw BaseExc();
        }
    }



    catch(BaseExc& exc) {
        std::cout << "Intercept BaseExc" << std::endl;
    }
    return 0;
}
```

## Result Information

**Group:** C++ Exception
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `EXCP_CAUGHT_BY_VALUE`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Exception handler hidden by previous handler

catch statement is not reached because of an earlier catch statement for the same exception

## Description

This defect occurs when a catch statement is not reached because a previous catch statement handles the exception.

For instance, a catch statement accepts an object of a class my_exception and a later catch statement accepts one of the following:

- An object of the my_exception class.
- An object of a class derived from the my_exception class.

### Risk

Because the catch statement is not reached, it is effectively dead code.

### Fix

One possible fix is to remove the redundant catch statement.

Another possible fix is to reverse the order of catch statements. Place the catch statement that accepts the derived class exception before the catch statement that accepts the base class exception.

## Examples

### catch Statement Hidden by Previous Statement

```
#include <new>

extern void print_str(const char* p);
extern void throw_exception();

void func() {
    try {
        throw_exception();
    }
    catch(std::exception& exc) {
        print_str(exc.what());
    }

    catch(std::bad_alloc& exc) {
        print_str(exc.what());
    }
}
```

In this example, the second catch statement accepts a std::bad_alloc object. Because the std::bad_alloc class is derived from a std::exception class, the second catch statement is hidden by the previous catch statement that accepts a std::exception object.

The defect appears on the parameter type of the `catch` statement. To find which `catch` statement hides the current `catch` statement:

1. On the **Source** pane, right-click the keyword `catch` and select **Search For** `"catch"`**in Current Source File**.
2. On the **Search** pane, click each search result, proceeding backwards from the current `catch` statement. Continue until you find the `catch` statement that hides the `catch` statement with the defect.

#### Correction — Reorder `catch` Statement

One possible correction is to place the `catch` statement with the derived class parameter first.

```
#include <new>

extern void print_str(const char* p);
extern void throw_exception();

void corrected_excphandlerhidden() {
    try {
        throw_exception();
    }

    catch(std::bad_alloc& exc) {
        print_str(exc.what());
    }
    catch(std::exception& exc) {
        print_str(exc.what());
    }
}
```

## Result Information

**Group:** C++ Exception
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `EXCP_HANDLER_HIDDEN`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Argument expression of throw statement might raise unexpected exception

The argument expression in a `throw` statement raises unexpected exceptions, leading to resource leaks and security vulnerabilities

## Description

This defect occurs when the argument expression of a throw statement might raise an exception. Expressions that can raise exceptions include:

- Functions that are specified as `noexcept(false)`
- Functions that contain one or more explicit `throw` statements
- Constructors that perform memory allocation operations
- Expressions that involve dynamic casting

### Risk

When raising an exception explicitly by using `throw` statements, the compiler first creates the expected exception by evaluating the argument of the throw statement, and then raises the expected exception. If an unexpected exception is raised when the compiler is creating the expected exception in a `throw` statement, the unexpected exception is propagated instead of the expected one. This unexpected exception might become an unhandled exception. Depending on your environment, the compiler might call `std::abort` to abnormally terminate the program execution without unwinding the stack when exceptions become unhandled, leading to resource leak and security vulnerabilities. Consider this code where a `throw` statement raises an explicit exception of class `myException`.

```
class myException{
    myException(){
        msg = new char[10];
        //...
    }
    //...
};

foo(){
    try{
        //..
        throw myException();
    }
    catch(myException& e){
        //...
    }
}
```

During construction of the temporary `myException` object, the `new` operator can raise a `bad_alloc` exception. In such a case, the `throw` statement raises a `bad_alloc` exception, instead of `myException`. Because `myException` was the expected exception, the catch block is incompatible with `bad_alloc`. The `bad_alloc` exception becomes an unhandled exception. It might cause the program to abort abnormally without unwinding the stack, leading to resource leak and security vulnerabilities.

#### Fix

Avoid using expressions that might raise exceptions as argument in a `throw` statement.

## Examples

### Use of Functions Specified as `noexcept(false)` in `throw` Statements

```
int f_throw() noexcept(false);
int foo(){
    try{
        //...
        throw f_throw();
    }
    catch(...){
        //...
    }
}
```

In this example, the function `f_throw()` is specified as `noexcept(false)`. If an exception is raised in `f_throw()`, it can cause the program to terminate without unwinding the stack, resulting in resource leak and security vulnerabilities.

#### Correction – Use Functions Specified As `noexcept(true)` as Argument Expression of `throw` Statements

One possible correction is to use functions that do not raise exceptions in argument expression of a `throw` statement. These functions are specified as `noexcept(true)`.

```
int f_throw() noexcept(true);
int foo(){
    try{
        //...
        throw f_throw();
    }
    catch(...){
        //...
    }
}
```

### Use of Constructors with Dynamic Memory Allocation in `throw` Statements

```
class WithDynamicAlloc {
public:
    WithDynamicAlloc(int n) {
        m_data = new int[n];
    }
    ~WithDynamicAlloc() {
        delete[] m_data;
    }
private:
    int* m_data;
};
int foo(){
    try{
        //...
        throw WithDynamicAlloc(10);
    }
    catch(WithDynamicAlloc& e){
        //...
    }
}
```

In this example, the constructor of the object `WithDynamicAlloc` performs a dynamic memory allocation operation. This constructor might raise an exception such as `bad_alloc` that might cause the program to terminate without unwinding the stack, resulting in resource leak and security vulnerabilities.

#### Correction – Use Constructors that Does Not Allocate Memory as Argument Expression of `throw` Statements

One possible correction is to use objects that do not require dynamic memory allocation as exception objects.

```
class WithoutDynamicAlloc {
public:
    WithoutDynamicAlloc(int n) : m_data(n){
    }
    ~WithoutDynamicAlloc() {
    }
private:
    int m_data;
};
int foo(){
    try{
        //...
        throw WithoutDynamicAlloc(10);
    }
    catch(WithoutDynamicAlloc& e){
        //...
    }
}
```

### Use of Functions That Might Raise Exceptions in `throw` Statements

```
int MightThrow(bool b) {
    if (b) {
        throw 2.1;
    }
    return 42;
}
int foo(){
    try{
        //...
        throw MightThrow(false);
        throw MightThrow(true);
    }
    catch(int e){
        //...
    }
}
```

In this example, the function `MightThrow()` raises an exception depending on the input `b`. Because Polyspace analyzes functions statically, it assumes that `MightThrow()` raises exceptions regardless of input, and raises a violation of this checker on both `throw MightThrow(true)` and `throw MightThrow(false)`.

**Correction – Use Comments To Justify Result**

Because `MightThrow(false)` does not raise an exception, a possible correction is to use comments to justify the statement `throw MightThrow(false)`.See“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”

```
int MightThrow(bool b) {
    if (b) {
        throw 2.1;
    }
    return 42;
}
int foo(){
    try{
        //...
        throw MightThrow(false);// polyspace DEFECT:THROW_THROWS [Justified:Low] "Does not Throw"
        throw MightThrow(true);
    }
    catch(int e){
        //...
    }
}
```

## Result Information

**Group:** C++ Exception
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `THROW_ARGUMENT_EXPRESSION_THROWS`
**Impact:** High

# Version History

**Introduced in R2020b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Noexcept function might exit with an exception

Functions specified as noexcept, noexcept(true) or noexcept(<true condition>) exits with an exception, which causes abnormal termination of program execution, leading to resource leak and security vulnerability

## Description

This defect occurs when a callable entity that is specified by using noexcept, noexcept(true), or noexcept(<true condition>) might exit with an exception.

When a callable entity invokes other callable entities, Polyspace makes certain assumptions to calculate whether there might be unhandled exceptions.

- Functions: When a noexcept function calls another function, Polyspace checks whether the called function might raise an exception only if it is specified as noexcept(<false>).If the called function is specified as noexcept, Polyspace assumes that it does not raise an exception. Some standard library functions, such as the constructor of std::string, use pointers to functions to perform memory allocation, which might raise exceptions. Because these functions are not specified as noexcept(<false>), Polyspace does not flag a function that calls these standard library functions.
- External function: When a noexcept function calls an external function, Polyspace flags the function declaration if the external function is specified as noexcept(<false>).
- Virtual function: When a function calls a virtual function, Polyspace flags the function declaration if the virtual function is specified as noexcept(<false>) in a derived class. For instance, if a noexcept function calls a virtual function that is declared as noexcept(<true>) in the base class, and noexcept(<false>) in a subsequent derived class, Polyspace flags the declaration of the noexcept function.
- Pointers to function: When a noexcept function invokes a pointer to a function, Polyspace assumes that the pointer to function does not raise exceptions.

When analyzing whether a function raises unhandled exceptions, Polyspace ignores:

- Exceptions raised in destructors
- Exceptions raised in atexit() operations
- Compiler generated functions

Polyspace also ignores the dynamic context when checking for exceptions. For instance, a function might raise unhandled exceptions that arise only in certain dynamic contexts. Polyspace flags such a function even if the exception might not be raised.

### Risk

You can specify that a callable entity does not exit with an exception by specifying it as noexcept, noexcept(true), or noexcept(<true condition>). The compiler omits the exception handing process for noexcept entities. When such an entity exits with an exception,the compiler implicitly invokes std::terminate().

Depending on the hardware and software you use, the function std::terminate() might invoke std::abort() to abnormally abort the execution without unwinding the stack. If the stack is not unwound before program termination, then the destructors of the variables in the stack are not invoked, leading to resource leak and security vulnerabilities.

### Fix

Specify functions as `noexcept` or `noexcept(true)` only when you know the functions raise no exceptions. If you cannot determine the exception specification of a function, specify it by using `noexcept(false)`

## Examples

### Use of `noexcept` Functions That Might Raise Exceptions

```
#include <stdexcept>
#include <typeinfo>
void LibraryFunc();
void LibraryFunc_noexcept_false() noexcept(false);
void LibraryFunc_noexcept_true() noexcept(true);



void SpecFalseCT() noexcept
{
    try {
        LibraryFunc_noexcept_false();
    } catch (int &e) {
        LibraryFunc_noexcept_false();
    } catch (std::exception &e) {
    } catch (...) {
    }
}

class A {
public:
    virtual void f() {}
};

class B : A {
public:
    virtual void f() noexcept {}
};

class C : B {
public:
    virtual void f() noexcept {}
};

class D : A {
public:
    virtual void f() noexcept(false) { throw(2);}
};

void A1(A &a) noexcept {
    a.f();
}
```

```
void D2(D &d) noexcept {
    try {
        d.f();
    } catch (int i) {
    } catch (...) {
    }
}

void B2(B *b) noexcept {
    b->f();
}
template <class T>
T f_tp(T a) noexcept(sizeof(T)<=4)
{
    if (sizeof(T) >4 ) {
        throw std::runtime_error("invalid case");
    }
    return a;
}
void instantiate(void)
{
    f_tp<char>(1);
}
void f() noexcept {
    throw std::runtime_error("dead code");
}

void g() noexcept {
    f();
}
```

In this example, there are several `noexcept` functions. These functions invoke other callable entities like functions, external functions, and virtual functions.

- Polyspace flags the declaration of the function template `f_tp` even though the `throw` statement is not reached because Polyspace ignores dynamic context. Polyspace also analyzes only the instantiated templates in your code. For instance, if `f_tp` is not instantiated in the function `instantiate()`, Polyspace does not analyze the template.
- Polyspace flags the `noexcept` function `SpecFaleCT()` because this function calls the `noexcept(false)`external function `LibraryFunc_noexcept_false()` without encapsulating it in a `try-catch` block. Any exceptions raised by this call to the external function might raise an unhandled exception.
- Polyspace flags the declaration of the `noexcept` function `A1()` because this function might call the `noexcept(false)` function `D.f()` when the input parameter `a` is of class `D`. Depending on the class of the input parameter, the `noexcept` polymorphic function `A1()` might raise an unhandled exception.
- Polyspace flags the function `f()` because it is a `noexcept` function that uses `throw` to raise an unhandled exception. Polyspace does not flag the `noexcept` function `g()` even though it calls `f()` because `f()` is specified as `noexcept`.
- Polyspace doe not flag the `noexcept` function `D2()` even though it calls the `noexcept(false)` function `D.f()` because `D2()` handles the exceptions that might arise by using a `catch(...)` block.

#### Correction – Specify Functions as noexcept(false)

You can modify your code so that the functions that might exit with an exception are specified as `noexcept(false)`. Such functions can exit with an exceptions without abnormally terminating the program and this defect is not raised.

```
#include <stdexcept>
#include <typeinfo>
void LibraryFunc_noexcept_false() noexcept(false);
void SpecFalseCT() noexcept(false)
{
    try {
        LibraryFunc_noexcept_false();
    } catch (int &e) {
        LibraryFunc_noexcept_false();
    } catch (std::exception &e) {
    } catch (...) {
    }
}
```

The function `SpecFalseCT()` uses an external function that is specified as `noexcept(false)`. The function `SpecFalseCT()` is specified as `noexcept(false)` because it might exit with an exception. This function does not raise the defect.

#### Correction – Handle Exceptions in noexcept Functions

You can modify your code so that exceptions are handled within a `noexcept` function. If raised exceptions and handled within the function, then the `noexcept` function does not exit with an exception and this defect is not raised.

```
#include <stdexcept>
#include <typeinfo>

void f() noexcept(false) {
throw(2);
}
void CallerFunc() noexcept {
    try {
        f();
    } catch (int i) {
    } catch (...) {
    }
}
```

The `noexcept` function `CallerFunc()` calls `f()` which can raise exceptions. `CallerFunc()` has handlers to handle exceptions that might be raised and does not exit with an exception. This functions does not raise the defect.

#### Correction – Justify Defects Using Comments

You can justify raised defects using comments: Because Polyspace analyzes functions statically, it might raise this defect to flag exceptions that are in dead code. Use comments to justify defects if you think the exception might not be raised.

```
#include <stdexcept>
```

```
#include <typeinfo>
void MightThrow(unsigned int input) noexcept{// polyspace DEFECT:NOEXCEPT_FUNCTION_THROWS
    if(input<0)
        throw 1;
    //..

}
```

The `noexcept` function `MightThrow()` exits with an exception in a dynamic context that might not arise. For instance, the `unsigned int` input is nonnegative and the `throw` statement does not execute. Because Polyspace analyzes functions statically, it raises the defect on the `throw` statement. Justify the defect using a comment. See“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”

### Result Information

**Group:** C++ Exception
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `NOEXCEPT_FUNCTION_THROWS`
**Impact:** High

## Version History

**Introduced in R2020b**

### See Also

`Find defects (-checkers)`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Uncaught exception

An exception is raised from a function but it is not caught and handled

## Description

This defect occurs when a function that is called in `main()` raises an exception and the exception is not handled. Polyspace highlights the location in the function body where the unhandled exception is raised and flags the call to the function in `main()`. For instance:

```
void foo(){
    throw std::exception(); //Uncaught exception
}
int main(){
    foo(); //Defect
    return 1;
}
```

Polyspace does not raise this defect when an `std::bad_alloc` raised by a `new` operator remains unhandled.

### Risk

When an exception remains unhandled, the compiler might invoke the function `std::terminate()`, which terminates the program abruptly. The abrupt termination does not invoke any exit handlers, does not call the destructors of the constructed objects, and does not unwind the stack.

Exceptions that are unhandled might result in issues such as memory leaks, security vulnerability, and other unintended behaviors. Poorly designed exception handling process might make your program vulnerable to denial-of-service attacks.

### Fix

To fix this defect, design the exception handling in your code to handle expected and unexpected exceptions. Call functions that are not `noexcept` in `try` blocks. Handle the exceptions that these functions might raise by using matching `catch()` blocks. Include a `catch-all` block to handle unexpected exceptions.

## Examples

### Avoid Unhandled Exceptions

```
#include <exception>
#include <stdexcept>

int flag();

void foo() {
    if (flag()==0) {
        //....
        throw std::exception();
    }
}

void bar() {
    if (flag()!=0 & flag()!=1) {
        throw std::logic_error("Error");
    }
}



void fubar() {
    foo();
}

int main() {
    foo(); // Defect
    bar(); // Defect
    fubar(); // Defect

}
```

In this exception, the functions `foo()` and `bar()` raise exceptions that are not handled. The function `fubar()` raises an unhandled exception by calling `foo()`. These functions are invoked in `main()`. Because these functions raise exceptions that are unhandled and are called from the `main()` function, Polyspace flags the calls to these functions in `main`.

**Correction — Handle the exceptions**

To resolve this defect, handle the exceptions in your code. For instance, in the `main()` function, call the exception raising function in a `try` block and handle the exception by using a series of `catch()` blocks, including a `catch(...)` block.

```
#include <exception>
#include <stdexcept>

int flag();

void foo() {
    if (flag()==0) {
        //....
        throw std::exception();
    }
}

void bar() {
    if (flag()!=0 & flag()!=1) {
        throw std::logic_error("bla");
    }
}



void fubar() {
    foo();
}

int main() {
    try{
```

```
    foo(); // Defect
    bar(); // Defect
    fubar(); // Defect
    }catch(std::logic_error& e){
        //...
    }catch(std::exception& e){
        //...
    }catch(...){
        //..
    }

}
```

## Result Information

**Group:** C++ Exception
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `UNCAUGHT_EXCEPTION`
**Impact:** High

# Version History

**Introduced in R2022b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Programming Defects

# Partially duplicated code

A section of code is duplicated in other places with very minor changes

## Description

This defect occurs when a block of code is duplicated in multiple places with only slight changes.

The defect checker does not flag certain blocks as duplicates. For instance, the blocks of code considered almost duplicates have to typically consist of more than a certain number of lines. See also “Duplicate Code Detection in Polyspace Bug Finder” on page 32-6.

Sometimes, two blocks might appear as exact duplicates, but might be reported as partial duplicates because some of the variables in the duplicate sections have different data types. The event list below the result points to the lines containing such variables.

### Risk

Sections of code that do almost the same operations might require unnecessary additional maintenance. Duplicated code also increases the chances you will update the code in one place but forget to update the other. See also `Duplicated code` and `Possible copy-paste error`.

### Fix

Try to refactor the sections of code to reduce the duplication. For instance, identify parts of the code blocks that are duplicates of each other and refactor them into a dedicated function. You can then replace the duplicated code with calls to the dedicated function.

## Examples

### Partially Duplicate Blocks of Code

In this example, the two code blocks in the functions `updatePinOne()` and `updatePinTwo()` are partial duplicates of each other. The duplicate section of each block copies one structure `currentBoard` to another structure `backupBoard` field by field. The only difference is that a different member of the structure `currentBoard` is updated in each function.

The code duplication can make maintenance difficult. For instance, if the structure type `board` is updated with more fields, it can be easy to forget copying the new fields in one of the duplicate sections. This omission can lead to a partial structure copy.

```
typedef struct {
    int pin1;
    int pin2;
    int pin3;
    int pin4;
    int pin5;
    int pin6;
    int pin7;
    int pin8;
} board;
```

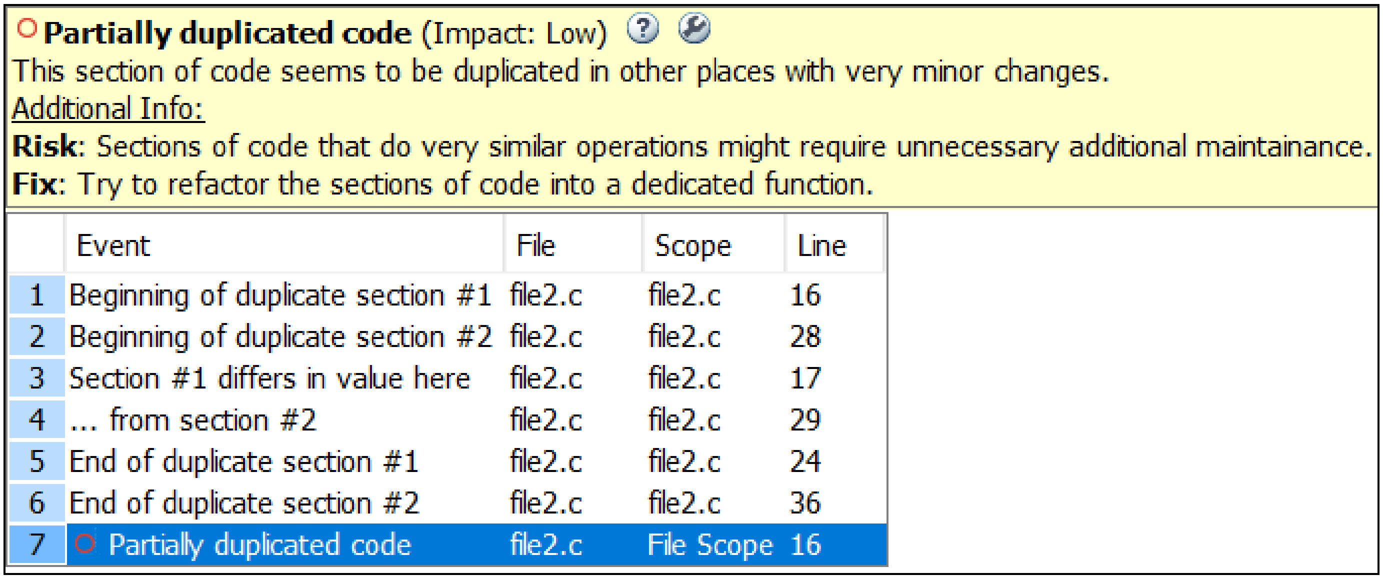
```
board currentBoard;
board backupBoard;

void updatePinOne(int newVal1) {
     backupBoard.pin1 = currentBoard.pin1; //Beginning of duplicate section #1
     currentBoard.pin1 = newVal1;
     backupBoard.pin2 = currentBoard.pin2;
     backupBoard.pin3 = currentBoard.pin3;
     backupBoard.pin4 = currentBoard.pin4;
     backupBoard.pin5 = currentBoard.pin5;
     backupBoard.pin6 = currentBoard.pin6;
     backupBoard.pin7 = currentBoard.pin7;
     backupBoard.pin8 = currentBoard.pin8; //End of duplicate section #1
}

void updatePinTwo(int newVal2) {
     backupBoard.pin1 = currentBoard.pin1; //Beginning of duplicate section #2
     currentBoard.pin2 = newVal2;
     backupBoard.pin2 = currentBoard.pin2;
     backupBoard.pin3 = currentBoard.pin3;
     backupBoard.pin4 = currentBoard.pin4;
     backupBoard.pin5 = currentBoard.pin5;
     backupBoard.pin6 = currentBoard.pin6;
     backupBoard.pin7 = currentBoard.pin7;
     backupBoard.pin8 = currentBoard.pin8; //End of duplicate section #2
}
```

The event list on the **Result Details** pane shows:

- The beginning and end of each duplicate block of code.
- The line where each block differs from the other.

  If two blocks are exactly identical except for the name of one variable and if the variable is used in multiple places, the event list shows only the first usage of the variable.

Click on an event to navigate to the corresponding location in the source code.



○ **Partially duplicated code** (Impact: Low)

This section of code seems to be duplicated in other places with very minor changes.

Additional Info:

**Risk**: Sections of code that do very similar operations might require unnecessary additional maintainance.

**Fix**: Try to refactor the sections of code into a dedicated function.

| | Event | File | Scope | Line |
|---|---|---|---|---|
| 1 | Beginning of duplicate section #1 | file2.c | file2.c | 16 |
| 2 | Beginning of duplicate section #2 | file2.c | file2.c | 28 |
| 3 | Section #1 differs in value here | file2.c | file2.c | 17 |
| 4 | ... from section #2 | file2.c | file2.c | 29 |
| 5 | End of duplicate section #1 | file2.c | file2.c | 24 |
| 6 | End of duplicate section #2 | file2.c | file2.c | 36 |
| 7 | ○ Partially duplicated code | file2.c | File Scope | 16 |

In the Polyspace user interface, you can see the duplicated code blocks side by side on the **Source** pane. See “Navigate in Separate Window”.

#### Correction – Refactor Code to Avoid Duplication

You can refactor the duplicate sections of code into a common function and replace the sections with calls to this function. In this example, the duplicate sections of the two blocks have been refactored into the `backupCurrentBoard()` function.

```
typedef struct {
    int pin1;
    int pin2;
    int pin3;
    int pin4;
    int pin5;
    int pin6;
    int pin7;
    int pin8;
} board;

board currentBoard;
board backupBoard;

void backupCurrentBoard(void) {
     backupBoard.pin1 = currentBoard.pin1;
     backupBoard.pin2 = currentBoard.pin2;
     backupBoard.pin3 = currentBoard.pin3;
     backupBoard.pin4 = currentBoard.pin4;
     backupBoard.pin5 = currentBoard.pin5;
     backupBoard.pin6 = currentBoard.pin6;
     backupBoard.pin7 = currentBoard.pin7;
     backupBoard.pin8 = currentBoard.pin8;
}

void updatePinOne(int newVal1) {
    backupCurrentBoard();
    currentBoard.pin1 = newVal1;
}

void updatePinTwo(int newVal2) {
    backupCurrentBoard();
    currentBoard.pin2 = newVal2;
}
```

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `ALMOST_DUPLICATED_CODE`
**Impact:** Low

# Version History

**Introduced in R2023a**

## See Also

`Find defects (-checkers)` | `Duplicated code` | `Possible copy-paste error`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Abnormal termination of exit handler

Exit handler function interrupts the normal execution of a program

## Description

This defect occurs when an exit handler itself calls another function that interrupts the program’s expected termination sequence and causes an abnormal exit.

- Exit handlers are functions designated for execution when a program terminates. These functions are first registered with specific functions such as `atexit`, (WinAPI) `_onexit`, or `at_quick_exit()`.
- Some functions that can cause abnormal exits are `exit`, `abort`, `longjmp`, or (WinAPI) `_onexit`.

### Risk

If your exit handler terminates your program, you can have undefined behavior. Abnormal program termination means other exit handlers are not invoked. These additional exit handlers may do additional clean up or other required termination steps.

### Fix

In inside exit handlers, remove calls to functions that prevent the exit handler from terminating normally.

## Examples

### Exit Handler With Call to exit

```
#include <stdlib.h>

volatile int some_condition = 1;
void demo_exit1(void)
{
    /* ... Cleanup code ... */
    return;
}
void exitabnormalhandler(void)
{
    if (some_condition)
    {
        /* Clean up */
        exit(0);
    }
    return;
}

int demo_install_exitabnormalhandler(void)
{

    if (atexit(demo_exit1) != 0) /* demo_exit1() performs additional cleanup */
    {
        /* Handle error */
    }
    if (atexit(exitabnormalhandler) != 0)
    {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

In this example, demo_install_exitabnormalhandler registers two exit handlers, demo_exit1 and exitabnormalhandler. Exit handlers are invoked in the reverse order of which they are registered. When the program ends, exitabnormalhandler runs, then demo_exit1. However, exitabnormalhandler calls exit interrupting the program exit process. Having this exit inside an exit handler causes undefined behavior because the program is not finished cleaning up safely.

#### Correction — Remove exit from Exit Handler

One possible correction is to let your exit handlers terminate normally. For this example, exit is removed from exitabnormalhandler, allowing the exit termination process to complete as expected.

```
#include <stdlib.h>

volatile int some_condition = 1;
void demo_exit1(void)
{
    /* ... Cleanup code ... */
    return;
}
void exitabnormalhandler(void)
{
    if (some_condition)
    {
        /* Clean up */
        /* Return normally */
    }
    return;
}

int demo_install_exitabnormalhandler(void)
{

    if (atexit(demo_exit1) != 0) /* demo_exit1() continues clean up */
    {
        /* Handle error */
    }
    if (atexit(exitabnormalhandler) != 0)
    {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `EXIT_ABNORMAL_HANDLER`
**Impact:** Medium

## Version History

**Introduced in R2016b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Accessing object with temporary lifetime

Read or write operations on the object are undefined behavior

## Description

This defect occurs when you attempt to read from or write to an object with temporary lifetime that is returned by a function call. In a structure or union returned by a function, and containing an array, the array members are temporary objects. The lifetime of temporary objects ends:

- When the full expression or full declarator containing the call ends, as defined in the C11 Standard.
- After the next sequence point, as defined in the C90 and C99 Standards. A sequence point is a point in the execution of a program where all previous evaluations are complete and no subsequent evaluation has started yet.

For C++ code, **Accessing object with temporary lifetime** raises a defect only when you write to an object with a temporary lifetime.

If the temporary lifetime object is returned by address, no defect is raised.

### Risk

Modifying objects with temporary lifetime is undefined behavior and can cause abnormal program termination and portability issues.

### Fix

Assign the object returned from the function call to a local variable. The content of the temporary lifetime object is copied to the variable. You can now modify it safely.

## Examples

### Modifying Temporary Lifetime Object Returned by Function Call

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

/* func_temp() returns a struct value containing
* an array with a temporary lifetime.
```

```
*/
int func(void) {

/*Writing to temporary lifetime object is
 undefined behavior
 */
    return ++(func_temp().a[0]);
}

void main(void) {
    (void)func();
}
```

In this example, `func_temp()` returns by value a structure with an array member `a`. This member has temporary lifetime. Incrementing it is undefined behavior.

**Correction — Assign Returned Value to Local Variable Before Writing**

One possible correction is to assign the return of the call to `func_temp()` to a local variable. The content of the temporary object `a` is copied to the variable, which you can safely increment.

```
 #include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

int func(void) {

/* Assign object returned by function call to
 *local variable
 */
    struct S_Array s = func_temp();

/* Local variable can safely be
 *incremented
 */
    ++(s.a[0]);
    return s.a[0];
}

void main(void) {
    (void)func();
}
```

## Result Information

**Group:** Programming
**Language:** C | C++

**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `TEMP_OBJECT_ACCESS`
**Impact:** Low

## Version History

**Introduced in R2018a**

## See Also

`Find defects (-checkers)` | `Misuse of structure with flexible array member` | `Write without a further read`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Alternating input and output from a stream without flush or positioning call

Undefined behavior for input or output stream operations

## Description

This defect occurs when:

- You do not perform a flush or function positioning call between an output operation and a following input operation on a file stream in update mode.
- You do not perform a function positioning call between an input operation and a following output operation on a file stream in update mode.

### Risk

Alternating input and output operations on a stream without an intervening flush or positioning call is undefined behavior.

### Fix

Call `fflush()` or a file positioning function such as `fseek()` or `fsetpos()` between output and input operations on an update stream.

Call a file positioning function between input and output operations on an update stream.

## Examples

### Read After Write Without Intervening Flush

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";

void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
```

```
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Read operation after write without
    intervening flush. */
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
          (void)fclose(file);
          /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

In this example, the file `demo.txt` is opened for reading and appending. After the call to `fwrite()`, a call to `fread()` without an intervening flush operation is undefined behavior.

**Correction — Call `fflush()` Before the Read Operation**

After writing data to the file, before calling `fread()`, perform a flush call.

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";


void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Buffer flush after write and before read */
    if (fflush(file) != 0)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

## Result Information

**Group:**Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `IO_INTERLEAVING`
**Impact:** Low

# Version History

**Introduced in R2017b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Assertion

Failed assertion statement

## Description

This defect occurs when you use an `assert`, and the asserted expression is or could be false.

**Note** Polyspace does not flag `assert(0)` as an assertion defect because these statements are commonly used to disable certain sections of code.

### Risk

Typically you use `assert` statements for functional testing in debug mode. An assertion failure found using static analysis indicates that the corresponding functional test would fail at run time.

### Fix

The fix depends on the root cause of the defect. For instance, the root cause can be unconstrained input from an external source that eventually led to the assertion failure.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Check Assertion on Unsigned Integer

```
#include <assert.h>

void asserting_x(unsigned int theta) {
    theta =+ 5;
    assert(theta < 0);
}
```

In this example, the `assert` function checks if the input variable, `theta`, is less than or equal to zero. The assertion fails because `theta` is an unsigned integer, so the value at the beginning of the function is at least zero. The += statement increases this positive value by five. Therefore, the range of `theta` is `[5..MAX_INT]`. `theta` is always greater than zero.

#### Correction — Change Assert Expression

One possible correction is to change the assertion expression. By changing the *less-than-or-equal-to* sign to a *greater-than-or-equal-to* sign, the assertion does not fail.

```
#include <assert.h>

void asserting_x(unsigned int theta) {
    theta =+ 5;
    assert(theta > 0);
}
```

#### Correction — Fix Code

One possible correction is to fix the code related to the assertion expression. If the assertion expression is true, fix your code so the assertion passes.

```
#include <assert.h>
#include <stdlib.h>

void asserting_x(int theta) {
    theta = -abs(theta);
    assert(theta < 0);
}
```

### Asserting Zero

```
#include <assert.h>

#define FLAG 0

int main(void){
    int i_test_z = 0;
    float f_test_z = (float)i_test_z;

    assert(i_test_z);
    assert(f_test_z);
    assert(FLAG);

    return 0;
}
```

In this example, Polyspace does not flag `assert(FLAG)` as a violation because a macro defines `FLAG` as `0`. The Polyspace Bug Finder assertion checker does not flag assertions with a constant zero parameter, `assert(0)`. These types of assertions are commonly used as dynamic checks during runtime. By inserting `assert(0)`, you indicate that the program must not reach this statement during run time, otherwise the program crashes.

However, the assertion checker does flag failed assertions caused by a variable value equal to zero, as seen in the example with `assert(i_test_z)` and `assert(f_test_z)`.

### Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `ASSERT`
**Impact:** High

## Version History

**Introduced in R2013b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Bad file access mode or status

Access mode argument of function in `fopen` or `open` group is invalid

## Description

This defect occurs when you use functions in the `fopen` or `open` group with invalid or incompatible file access modes, file creation flags, or file status flags as arguments. For instance, for the `open` function, examples of valid:

- Access modes include `O_RDONLY`, `O_WRONLY`, and `O_RDWR`
- File creation flags include `O_CREAT`, `O_EXCL`, `O_NOCTTY`, and `O_TRUNC`.
- File status flags include `O_APPEND`, `O_ASYNC`, `O_CLOEXEC`, `O_DIRECT`, `O_DIRECTORY`, `O_LARGEFILE`, `O_NOATIME`, `O_NOFOLLOW`, `O_NONBLOCK`, `O_NDELAY`, `O_SHLOCK`, `O_EXLOCK`, `O_FSYNC`, `O_SYNC` and so on.

The defect can occur in the following situations.

| Situation | Risk | Fix |
|---|---|---|
| You pass an empty or invalid access mode to the `fopen` function.<br>According to the ANSI C standard, the valid access modes for `fopen` are:<br>• `r`,`r+`<br>• `w`,`w+`<br>• `a`,`a+`<br>• `rb`, `wb`, `ab`<br>• `r+b`, `w+b`, `a+b`<br>• `rb+`, `wb+`, `ab+` | `fopen` has undefined behavior for invalid access modes.<br>Some implementations allow extension of the access mode such as:<br>• GNU: `rb+cmxe,ccs=utf`<br>• Visual C++: `a+t`, where `t` specifies a text mode.<br>However, your access mode string must begin with one of the valid sequences. | Pass a valid access mode to `fopen`. |
| You pass the status flag `O_APPEND` to the `open` function without combining it with either `O_WRONLY` or `O_RDWR`. | `O_APPEND` indicates that you intend to add new content at the end of a file. However, without `O_WRONLY` or `O_RDWR`, you cannot write to the file.<br>The `open` function does not return -1 for this logical error. | Pass either `O_APPEND\|O_WRONLY` or `O_APPEND\|O_RDWR` as access mode. |

| Situation | Risk | Fix |
|---|---|---|
| You pass the status flags O_APPEND and O_TRUNC together to the open function. | O_APPEND indicates that you intend to add new content at the end of a file. However, O_TRUNC indicates that you intend to truncate the file to zero. Therefore, the two modes cannot operate together.<br>The open function does not return -1 for this logical error. | Depending on what you intend to do, pass one of the two modes. |
| You pass the status flag O_ASYNC to the open function. | On certain implementations, the mode O_ASYNC does not enable signal-driven I/O operations. | Use the fcntl(pathname, F_SETFL, O_ASYNC); instead. |

### Fix

The fix depends on the function and the flags used. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Invalid Access Mode with fopen

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "rw");
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

In this example, the access mode rw is invalid. Because r indicates that you open the file for reading and w indicates that you create a new file for writing, the two access modes are incompatible.

#### Correction — Use Either r or w as Access Mode

One possible correction is to use the access mode corresponding to what you intend to do.

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "w");
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `BAD_FILE_ACCESS_MODE_STATUS`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Call through non-prototyped function pointer

Function pointer declared without its type or number of parameters causes unexpected behavior

## Description

This defect occurs when a function without a complete prototype is called using a function pointer.

A function prototype specifies the type and number of parameters.

### Risk

Arguments passed to a function without a prototype might not match the number and type of parameters of the function definition, which can cause undefined behavior. If the parameters are restricted to a subset of their type domain, arguments from untrusted sources can trigger vulnerabilities in the called function.

### Fix

Before calling the function through a pointer, provide a function prototype.

## Examples

### Argument Does Not Match Parameter Restriction

```
#include <stdio.h>
#include <limits.h>
#define SIZE2 2

typedef void (*func_ptr)();
extern int getchar_wrapper(void);
extern void restricted_int_sink(int i);
/* Integer value restricted to
range [-1, 255] */
extern void restricted_float_sink(double i);
/* Double value restricted to > 0.0 */



func_ptr generic_callback[SIZE2] =
{
    (func_ptr)restricted_int_sink,
    (func_ptr)restricted_float_sink
};

void func(void)
{
    int ic;
    ic = getchar_wrapper();
    /* Wrong index used for generic_callback.
    Negative 'int' passed to restricted_float_sink. */
    (*generic_callback[1])(ic);
}
```

In this example, a call through `func_ptr` passes `ic` as an argument to function `generic_callback[1]`. The type of `ic` can have negative values, while the parameter of `generic_callback[1]` is restricted to float values greater than `0.0`. Typically, compilers and static analysis tools cannot perform type checking when you do not provide a pointer prototype.

**Correction — Provide Prototype of Pointer to Function**

Pass the argument `ic` to a function with a parameter of type `int`, by using a properly prototyped pointer.

```
#include <stdio.h>
#include <limits.h>
#define SIZE2 2

typedef void (*func_ptr_proto)(int);
extern int getchar_wrapper(void);
extern void restricted_int_sink(int i);
/* Integer value restricted to
range [-1, 255] */
extern void restricted_float_sink(double i);
/* Double value restricted to > 0.0 */

func_ptr_proto generic_callback[SIZE2] =
{
    (func_ptr_proto)restricted_int_sink,
    (func_ptr_proto)restricted_float_sink
};

void func(void)
{
    int ic;
    ic = getchar_wrapper();
    /* ic passed to function through
properly prototyped pointer. */
    (*generic_callback[0])(ic);
}
```

## Result Information

**Group:** Programming
**Language:** C
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `UNPROTOTYPED_FUNC_CALL`
**Impact:** Medium

# Version History

**Introduced in R2017b**

## See Also

`Declaration mismatch` | `Unreliable cast of function pointer` | `Find defects (-checkers)`

**Topics**

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Call to memset family with unintended value

memset or wmemset used with possibly incorrect arguments

## Description

This defect occurs when Polyspace Bug Finder detects a use of the memset or wmemset function with possibly incorrect arguments.

void *memset (void *ptr, int value, size_t num) fills the first num bytes of the memory block that ptr points to with the specified value. If the argument value is incorrect, the memory block is initialized with an unintended value.

The unintended initialization can occur in the following cases.

| Issue | Risk | Possible Fix |
|---|---|---|
| The second argument is '0' instead of 0 or '\0'. | The ASCII value of character '0' is 48 (decimal), 0x30 (hexadecimal), 069 (octal) but not 0 (or '\0') . | If you want to initialize with '0', use one of the ASCII values. Otherwise, use 0 or '\0'. |
| The second and third arguments are probably reversed. For instance, the third argument is a literal and the second argument is not a literal. | If the order is reversed, a memory block of unintended size is initialized with incorrect arguments. | Reverse the order of the arguments. |
| The second argument cannot be represented in a byte. | If the second argument cannot be represented in a byte, and you expect each byte of a memory block to be filled with that argument, the initialization does not occur as intended. | Apply a bit mask to the argument to produce a wrapped or truncated result that can be represented in a byte. When you apply a bit mask, make sure that it produces an expected result.<br>For instance, replace memset(a, -13, sizeof(a)) with memset(a, (-13) & 0xFF, sizeof(a)). |

### Fix

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.

- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Value Cannot Be Represented in a Byte

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE];
    int c = -2;
    memset(buf, (char)c, sizeof(buf));
}
```

In this example, `(char)c` cannot be represented in a byte.

#### Correction — Apply Cast

One possible correction is to apply a cast so that the result can be represented in a byte. Check that the result of the cast is an acceptable initialization value. In this correction, Polyspace does not raise this defect. The cast from signed `int` to `unsigned char` is contrary to best practices and Polyspace raises the defect `Sign change integer conversion overflow`.

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE   ];
    int c = -2;
    memset(buf, (unsigned char)c, sizeof(buf));// Might Overflow
}
```

#### Correction — Avoid Using memset

One possible correction is to reserve the use of `memset` only for setting or clearing all bits in a buffer. For instance, in this code, `memset` is called to clear the bits of the character array `buf`.

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE   ];
    int c = -2;
    memset(buf, 0, sizeof(buf));//Compliant
    /* After clearing buf, use it in operations*/
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `MEMSET_INVALID_VALUE`
**Impact:** Low

## Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)` | `Use of memset with size argument zero`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Character value absorbed into EOF

Data type conversion makes a valid character value same as End-of-File (EOF)

## Description

This defect occurs when you perform a data type conversion that makes a valid character value indistinguishable from `EOF` (End-of-File). Bug Finder flags the defect in one of the following situations:

- *End-of-File*: You perform a data type conversion such as from `int` to `char` that converts a non-EOF character value into `EOF`.

  ```
  char ch = (char)getchar()
  ```

  You then compare the result with EOF.

  ```
  if((int)ch == EOF)
  ```

  The conversion can be explicit or implicit.
- *Wide End-of-File*: You perform a data type conversion that can convert a non-WEOF wide character value into WEOF, and then compare the result with WEOF.

### Risk

The data type `char` cannot hold the value `EOF` that indicates the end of a file. Functions such as `getchar` have return type `int` to accommodate `EOF`. If you convert from `int` to `char`, the values `UCHAR_MAX` (a valid character value) and `EOF` get converted to the same value -1 and become indistinguishable from each other. When you compare the result of this conversion with `EOF`, the comparison can lead to false detection of `EOF`. This rationale also applies to wide character values and `WEOF`.

### Fix

Perform the comparison with `EOF` or `WEOF` before conversion.

## Examples

### Return Value of `getchar` Converted to `char`

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

char func(void)
{
    char ch;
    ch = getchar();
    if (EOF == (int)ch) {
        fatal_error();
    }
```

```
    return ch;
}
```

In this example, the return value of `getchar` is implicitly converted to `char`. If `getchar` returns `UCHAR_MAX`, it is converted to -1, which is indistinguishable from EOF. When you compare with EOF later, it can lead to a false positive.

**Correction — Perform Comparison with EOF Before Conversion**

One possible correction is to first perform the comparison with EOF, and then convert from `int` to `char`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

char func(void)
{
    int i;
    i = getchar();
    if (EOF == i) {
        fatal_error();
    }
    else {
        return (char)i;
    }
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `CHAR_EOF_CONFUSED`
**Impact:** High

# Version History

**Introduced in R2017a**

## See Also

`Find defects (-checkers)` | `Invalid use of standard library integer routine` | `Returned value of a sensitive function not checked` | `Errno not checked` | `Misuse of sign-extended character value`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Copy of overlapping memory

Source and destination arguments of a copy function have overlapping memory

## Description

This defect occurs when there is a memory overlap between the source and destination argument of a copy function such as `memcpy` or `strcpy`. For instance, the source and destination arguments of `strcpy` are pointers to different elements in the same string.

### Risk

If there is memory overlap between the source and destination arguments of copy functions, according to C standards, the behavior is undefined.

### Fix

Determine if the memory overlap is what you want. If so, find an alternative function. For instance:

- If you are using `memcpy` to copy values from one memory location to another, use `memmove` instead of `memcpy`.
- If you are using `strcpy` to copy one string to another, use `memmove` instead of `strcpy`, as follows:

  ```
  s = strlen(source);
  memmove(destination, source, s + 1);
  ```

  `strlen` determines the string length without the null terminator. Therefore, you must move `s+1` bytes instead of `s` bytes.

## Examples

### Overlapping Copy

```
#include <string.h>

char str[] = {"ABCDEFGH"};

void my_copy() {
    strcpy(&str[0],(const char*)&str[2]);
}
```

In this example, because the source and destination argument are pointers to the same string `str`, there is memory overlap between their allowed buffers.

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `OVERLAPPING_COPY`
**Impact:** Medium

## Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)` | `Overlapping assignment`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Possible copy-paste error

A section of code is duplicated in other places with exactly one minor change

## Description

This defect occurs when a block of code is duplicated in multiple places with exactly one minor change.

The defect checker does not flag certain blocks as duplicates. For instance, the blocks of code considered almost duplicates have to typically consist of more than a certain number of lines. See also “Duplicate Code Detection in Polyspace Bug Finder” on page 32-6.

### Risk

Sections of code that are exact duplicates apart from one minor change might indicate a programming error. You might have duplicated a section of code and later updated one of the duplicates but forgot to update the other.

Duplicated code also require unnecessary additional maintenance. See also `Almost duplicated code` and `Duplicated code`.

### Fix

See if the one minor change between the two duplicated blocks is the result of a programming error. Fix the error.

In the long run, try to refactor the sections of code to reduce the duplication. For instance, identify parts of the code blocks that are duplicates of each other and refactor them into a dedicated function. You can then replace the duplicated code with calls to the dedicated function.

## Examples

### Duplicate Blocks of Code With One Minor Change

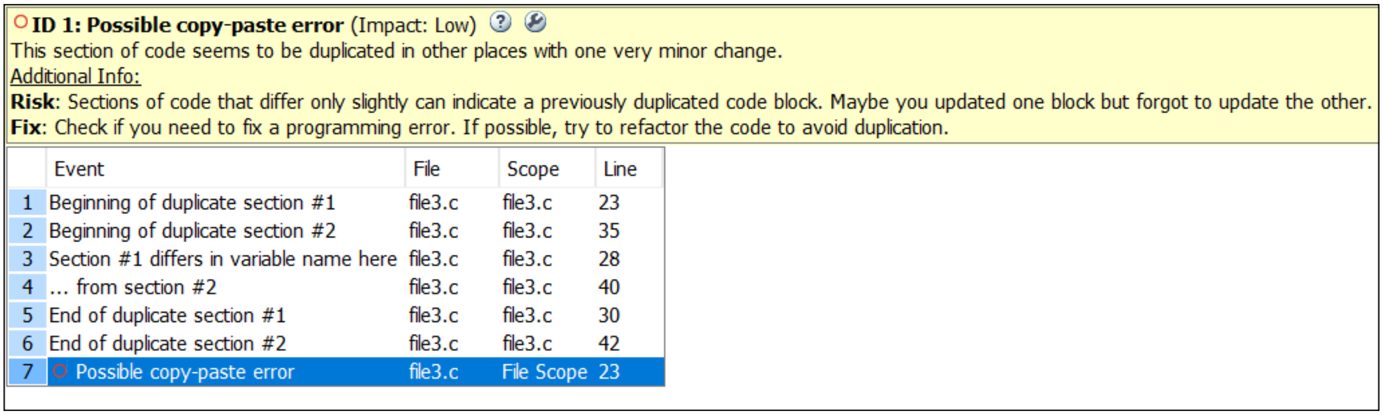
In this example, the functions `getReadings()` and `isAllOk()` contain a duplicate block of code with only one minor change. This one difference strongly indicates the possibility of a copy-paste error. It is possible that the logic for calculating the variables `currentReading1` to `currentReading8` is intended to be the same in the both functions. The logic was initially the same, but possibly updated later in one function but not in the other.

```
int pin1;
int pin2;
int pin3;
int pin4;
int pin5;
int pin6;
int pin7;
int pin8;
int currentReading1;
int currentReading2;
int currentReading3;
int currentReading4;
int currentReading5;
int currentReading6;
int currentReading7;
int currentReading8;

int allOk;

void getReadings() {
    currentReading1 = pin1; //Beginning of duplicate section #1
    currentReading2 = pin2 - pin1;
    currentReading3 = pin2 + pin3;
    currentReading4 = pin4;
    currentReading5 = pin5;
    currentReading6 = pin6 - pin5 + pin2; //Minor change here
    currentReading7 = pin7;
    currentReading8 = pin8; //End of duplicate section #1

}

void isAllOk() {
    currentReading1 = pin1; //Beginning of duplicate section #2
    currentReading2 = pin2 - pin1;
    currentReading3 = pin2 + pin3;
    currentReading4 = pin4;
    currentReading5 = pin5;
    currentReading6 = pin6 - pin5 + pin1; //Minor change here
    currentReading7 = pin7;
    currentReading8 = pin8; //End of duplicate section #2

    if(currentReading1 > currentReading2
       && currentReading3 > currentReading4
       && currentReading5 > currentReading6
       && currentReading7 > currentReading8)
       allOk = 1;
}
```

The event list on the **Result Details** pane shows:

- The beginning and end of each duplicate block of code.
- The line where each block differs from the others.

Click on an event to navigate to the corresponding location in the source code.



○ **ID 1: Possible copy-paste error** (Impact: Low)
This section of code seems to be duplicated in other places with one very minor change.
Additional Info:
**Risk**: Sections of code that differ only slightly can indicate a previously duplicated code block. Maybe you updated one block but forgot to update the other.
**Fix**: Check if you need to fix a programming error. If possible, try to refactor the code to avoid duplication.

| | Event | File | Scope | Line |
|---|---|---|---|---|
| 1 | Beginning of duplicate section #1 | file3.c | file3.c | 23 |
| 2 | Beginning of duplicate section #2 | file3.c | file3.c | 35 |
| 3 | Section #1 differs in variable name here | file3.c | file3.c | 28 |
| 4 | ... from section #2 | file3.c | file3.c | 40 |
| 5 | End of duplicate section #1 | file3.c | file3.c | 30 |
| 6 | End of duplicate section #2 | file3.c | file3.c | 42 |
| 7 | ○ Possible copy-paste error | file3.c | File Scope | 23 |

In the Polyspace user interface, you can see the duplicated code blocks side by side on the **Source** pane. See “Navigate in Separate Window”.

**Correction – Fix Errors and Refactor Code to Avoid Duplication**

If the issue flagged indicates a possible copy-paste error such as a forgotten update, fix the programming error. If possible, refactor your code to avoid the code duplication.

In this example, the function `isAllOk()` has been refactored so that it calls the function `getReadings()` to calculate variables `currentReading1` to `currentReading8`. This refactoring eliminates chances of a copy-paste error.

```
int pin1;
int pin2;
int pin3;
int pin4;
int pin5;
int pin6;
int pin7;
int pin8;


int currentReading1;
int currentReading2;
int currentReading3;
int currentReading4;
int currentReading5;
int currentReading6;
int currentReading7;
int currentReading8;

int allOk;

void getReadings() {
    currentReading1 = pin1;
    currentReading2 = pin2 - pin1;
    currentReading3 = pin2 + pin3;
    currentReading4 = pin4;
    currentReading5 = pin5;
    currentReading6 = pin6 - pin5 + pin2;
    currentReading7 = pin7;
    currentReading8 = pin8;
}

void isAllOk() {
    getReadings();

    if(currentReading1 > currentReading2
       && currentReading3 > currentReading4
       && currentReading5 > currentReading6
       && currentReading7 > currentReading8)
       allOk = 1;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `COPY_PASTE_ERROR`
**Impact:** Low

# Version History

**Introduced in R2023a**

## See Also

`Find defects (-checkers)` | `Partially duplicated code` | `Duplicated code`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Declaration mismatch

Mismatch between function or variable declarations

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.

This defect occurs when a function or variable declaration does not match other instances of the function or variable.

### Risk

When a mismatch occurs between two variable declarations in different compilation units, a typical linker follows an algorithm to pick one declaration for the variable. If you expect a variable declaration that is different from the one chosen by the linker, you can see unexpected results when the variable is used.

A similar issue can occur with mismatch in function declarations.

### Fix

The fix depends on the type of declaration mismatch. If both declarations indeed refer to the same object, use the same declaration. If the declarations refer to different objects, change the names of the one of the variables. If you change a variable name, remember to make the change in all places that use the variable.

Sometimes, declaration mismatches can occur because the declarations are affected by previous preprocessing directives. For instance, a declaration occurs in a macro, and the macro is defined on one inclusion path but undefined in another. These declaration mismatches can be tricky to debug. Identify the divergence between the two inclusion paths and fix the conflicting macro definitions.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Inconsistent Declarations in Two Files

*file1.c*

```
int foo(void) {
    return 1;
}
```

*file2.c*

```
double foo(void);

int bar(void) {
    return (int)foo();
}
```

In this example, *file1.c* declares `foo()` as returning an integer. In *file2.c*, `foo()` is declared as returning a double. This difference might cause a compile failure. Polyspace raises a defect on the second instance of `foo` in *file2*.

#### Correction — Align the Function Return Values

One possible correction is to change the function declarations so that they match. In this example, by changing the declaration of `foo` in *file2.c* to match *file1.c*, the defect is fixed.

*file1.c*

```
int foo(void) {
    return 1;
}
```

*file2.c*

```
int foo(void);

int bar(void) {
    return foo();
}
```

### Inconsistent Structure Alignment

*test1.c*

```
#include<stdio.h>
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

void func2();

int main(){
    square.side=1;
    circle.radius=1;

    func2();
    return 0;
}
```

*test2.c*

```
#include<stdio.h>
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

void func2(){
    printf("%d\n", square.side);
    printf("%d\n", circle.radius);
}
```

*circle.h*

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;
```

*square.h*

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

In this example, a declaration mismatch defect is raised on `square` in *test2.c* because Polyspace infers that `square` in *square.h* does not have the same alignment as `square` in *test2.c*. This error occurs because the `#pragma pack(1)` statement in *circle.h* declares specific alignment. In *test2.c*, *circle.h* is included before *square.h*. Therefore, the `#pragma pack(1)` statement from *circle.h* is not reset to the default alignment after the `aCircle` structure. Because of this omission, *test2.c* infers that the `aSquare square` structure also has an alignment of `1` byte. This defect might cause a compilation failure.

### Correction — Close Packing Statements

One possible correction is to reset the structure alignment after the `aCircle` struct declaration. For the GNU or Microsoft Visual compilers, fix the defect by adding a `#pragma pack()` statement at the end of *circle.h*.

*test1.c*

```
#include<stdio.h>
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

void func2();

int main(){
    square.side=1;
    circle.radius=1;

    func2();
    return 0;
}
```

*test2.c*

```
#include<stdio.h>
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

void func2(){
    printf("%d\n", square.side);
    printf("%d\n", circle.radius);
}
```

*circle.h*

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;

#pragma pack()
```

*square.h*

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

Other compilers require different `#pragma pack` syntax. For your syntax, see the documentation for your compiler.

### Correction — Use the `Ignore pragma pack directives` Option

One possible correction is to add the `Ignore pragma pack directives` option to your Bug Finder analysis. If you want the structure alignment to change for each structure, and you do not want to see this **Declaration mismatch** defect, use this correction.

1. On the Configuration pane, select the **Advanced Settings** pane.
2. In the **Other** box, enter `-ignore-pragma-pack`.
3. Rerun your analysis.

   The **Declaration mismatch** defect is resolved.

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `DECL_MISMATCH`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Ignore pragma pack directives (-ignore-pragma-pack)` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Duplicated code

A section of code is duplicated in other places

## Description

This defect occurs when a block of code is duplicated in multiple places.

The defect checker does not flag certain blocks as duplicates. For instance, the blocks of code considered almost duplicates have to typically consist of more than a certain number of lines. See also “Duplicate Code Detection in Polyspace Bug Finder” on page 32-6.

### Risk

Sections of code that do the same operations require unnecessary additional maintenance. Duplicated code also increases the chances you will update the code in one place but forget to update the other. See also `Almost duplicated code` and `Possible copy-paste error`.

### Fix

Refactor the sections of code into a dedicated function. In other words, if two blocks of code are duplicates of each other, write a new function containing this code block and replace the existing blocks with calls to the new function.

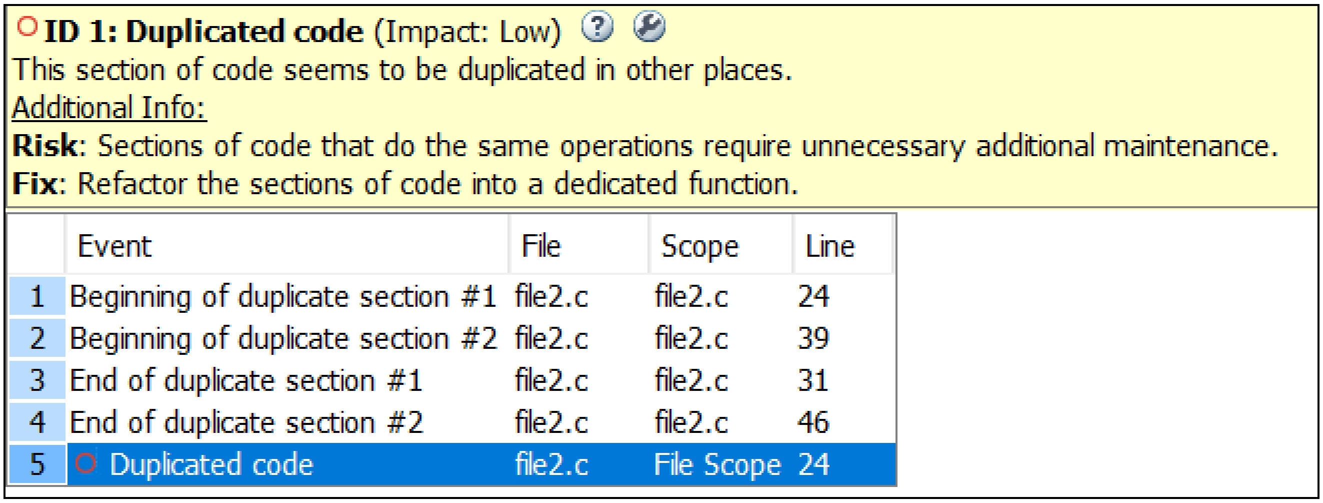
## Examples

### Duplicate Blocks of Code

This example contains two blocks of code that are exact duplicates of each other.

```
typedef struct {
    int pin1;
    int pin2;
    int pin3;
    int pin4;
    int pin5;
    int pin6;
    int pin7;
    int pin8;
} board;

board Aboard;
extern int getPinVal();

void setAllPins() {
     int val1 = getPinVal();
     int val2 = getPinVal();
     int val3 = getPinVal();
     int val4 = getPinVal();
     int val5 = getPinVal();
     int val6 = getPinVal();
     int val7 = getPinVal();
     int val8 = getPinVal();
     Aboard.pin1 = val1; //Beginning of duplicate section #1
     Aboard.pin2 = val2;
     Aboard.pin3 = val3;
     Aboard.pin4 = val4;
     Aboard.pin5 = val5;
     Aboard.pin6 = val6;
     Aboard.pin7 = val7;
     Aboard.pin8 = val8; //End of duplicate section #1

}

void resetAllPins() {
     int val1 = 0, val2 = 1, val3 = 0,
         val4 = 0, val5 = 1, val6 = 0,
         val7 = 0, val8 = 1;
     Aboard.pin1 = val1; //Beginning of duplicate section #2
     Aboard.pin2 = val2;
     Aboard.pin3 = val3;
     Aboard.pin4 = val4;
     Aboard.pin5 = val5;
     Aboard.pin6 = val6;
     Aboard.pin7 = val7;
     Aboard.pin8 = val8; //End of duplicate section #2
}
```

The event list on the **Result Details** pane shows the beginning and end of each block of code. Click on an event to navigate to the corresponding location in the source code.



○ **ID 1: Duplicated code** (Impact: Low)

This section of code seems to be duplicated in other places.

Additional Info:

**Risk**: Sections of code that do the same operations require unnecessary additional maintenance.

**Fix**: Refactor the sections of code into a dedicated function.

| | Event | File | Scope | Line |
|---|---|---|---|---|
| 1 | Beginning of duplicate section #1 | file2.c | file2.c | 24 |
| 2 | Beginning of duplicate section #2 | file2.c | file2.c | 39 |
| 3 | End of duplicate section #1 | file2.c | file2.c | 31 |
| 4 | End of duplicate section #2 | file2.c | file2.c | 46 |
| 5 | ○ Duplicated code | file2.c | File Scope | 24 |

In the Polyspace user interface, you can see the duplicated code blocks side by side on the **Source** pane. See “Navigate in Separate Window”.

**Correction – Refactor Duplicate Sections Into Common Function**

You can refactor the duplicate sections of code into a common function and replace the sections with calls to this function. In this example, the duplicate block has been refactored into the `init()` function.

```
typedef struct {
    int pin1;
    int pin2;
    int pin3;
    int pin4;
    int pin5;
    int pin6;
    int pin7;
    int pin8;
} board;

board Aboard;
extern int getPinVal();

void init(int val1, int val2, int val3, int val4,
          int val5, int val6, int val7, int val8) {
     Aboard.pin1 = val1;
     Aboard.pin2 = val2;
     Aboard.pin3 = val3;
     Aboard.pin4 = val4;
     Aboard.pin5 = val5;
     Aboard.pin6 = val6;
     Aboard.pin7 = val7;
     Aboard.pin8 = val8;
}

void setAllPins() {
     int val1 = getPinVal();
     int val2 = getPinVal();
     int val3 = getPinVal();
     int val4 = getPinVal();
     int val5 = getPinVal();
     int val6 = getPinVal();
     int val7 = getPinVal();
     int val8 = getPinVal();
     init(val1, val2, val3, val4, val5, val6, val7, val8);
}

void resetAllPins() {
     int val1 = 0, val2 = 1, val3 = 0,
         val4 = 0, val5 = 1, val6 = 0,
         val7 = 0, val8 = 1;
     init(val1, val2, val3, val4, val5, val6, val7, val8);
}
```

### Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `DUPLICATED_CODE`
**Impact:** Low

## Version History

**Introduced in R2023a**

## See Also

`Find defects (-checkers)` | `Possible copy-paste error` | `Partially duplicated code`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Environment pointer invalidated by previous operation

Call to setenv or putenv family function modifies environment pointed to by pointer

## Description

This defect occurs when you use the third argument of *main()* in a hosted environment to access the environment after an operation modifies the environment. In a hosted environment, many C implementations support the nonstandard syntax:

```
main (int argc, char *argv[], char *envp[])
```

A call to a setenv or putenv family function modifies the environment pointed to by *envp.

### Risk

When you modify the environment through a call to a setenv or putenv family function, the environment memory can potentially be reallocated. The hosted environment pointer is not updated and might point to an incorrect location. A call to this pointer can return unexpected results or cause an abnormal program termination.

### Fix

Do not use the hosted environment pointer. Instead, use global external variable environ in Linux, _environ or _wenviron in Windows, or their equivalent. When you modify the environment, these variables are updated.

## Examples

### Access Environment Through Pointer envp

```
#include <stdio.h>
#include <stdlib.h>

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

/* envp is from main function */
int func(char **envp)
{
    /* Call to setenv may cause environment
     *memory to be reallocated
     */
    if (setenv(("MY_NEW_VAR"),("new_value"),1) != 0)
    {
        /* Handle error */
        return -1;
    }
    /* envp not updated after call to setenv, and may
     *point to incorrect location.
     **/
    if (envp != ((void *)0)) {
        use_envp(envp);
/* No defect on second access to
*envp because defect already raised */
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func(envp);
    }
}
```

In this example, `envp` is accessed inside `func()` after a call to `setenv` that can reallocate the environment memory. `envp` can point to an incorrect location because it is not updated after `setenv` modifies the environment. No defect is raised when `use_envp()` is called because the defect is already raised on the previous line of code.

#### Correction — Use Global External Variable `environ`

One possible correction is to access the environment by using a variable that is always updated after a call to `setenv`. For instance, in the following code, the pointer `envp` is still available from `main()`, but the environment is accessed in `func()` through the global external variable `environ`.

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

int func(void)
{
    if (setenv(("MY_NEW_VAR"), ("new_value"),1) != 0) {
        /* Handle error */
        return -1;
    }
  /* Use global external variable environ
   *which is always updated after a call to setenv */

    if (environ != NULL) {
        use_envp(environ);
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func();
    }
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `INVALID_ENV_POINTER`
**Impact:** Medium

# Version History

**Introduced in R2018a**

## See Also

`Find defects (-checkers)` | `Misuse of return value from nonreentrant standard function` | `Modification of internal buffer returned from nonreentrant standard function`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Errno not reset

`errno` not reset before calling a function that sets `errno`

## Description

This defect occurs when you do not reset `errno` before calling a function that sets `errno` to indicate error conditions. However, you check `errno` for those error conditions after the function call.

### Risk

The `errno` is not clean and can contain values from a previous call. Checking `errno` for errors can give the false impression that an error occurred.

`errno` is set to zero at program startup but subsequently, `errno` is not reset by a C standard library function. You must explicitly set `errno` to zero when required.

### Fix

Before calling a function that sets `errno` to indicate error conditions, reset `errno` to zero explicitly.

## Examples

### `errno` Not Reset Before Call to `strtod`

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
                  {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

In this example, `errno` is not reset to 0 before the first call to `strtod`. Checking `errno` for 0 later can lead to a false positive.

#### Correction — Reset `errno` Before Call

One possible correction is to reset `errno` to 0 before calling `strtod`.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    errno = 0;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
              {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `MISSING_ERRNO_RESET`
**Impact:** High

# Version History

**Introduced in R2017a**

## See Also

`Find defects (-checkers)` | `Returned value of a sensitive function not checked` | `Misuse of errno` | `Errno not checked`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Floating point comparison with equality operators

Imprecise comparison of floating-point variables

## Description

This defect occurs when you use an equality (==) or inequality (!=) operation with floating-point numbers.

Polyspace does not raise a defect for an equality or inequality operation with floating-point numbers when:

- The comparison is between two float constants.

  ```
      float flt = 1.0;
      if (flt == 1.1)
  ```
- The comparison is between a constant and a variable that can take a finite, reasonably small number of values.

  ```
  float x;

  int rand = random();
  switch(rand) {
  case 1: x = 0.0; break;
  case 2: x = 1.3; break;
  case 3: x = 1.7; break;
  case 4: x = 2.0; break;
  default: x = 3.5; break; }
  …
  if (x==1.3)
  ```
- The comparison is between floating-point expressions that contain only integer values.

  ```
  float x = 0.0;
  for (x=0.0;x!=100.0;x+=1.0) {
  …
  if (random) break;
  }

  if (3*x+4==2*x-1)
  …
  if (3*x+4 == 1.3)
  ```
- One of the operands is `0.0`, unless you use the option flag `-detect-bad-float-op-on-zero`.

  ```
  /* Defect detected when
  you use the option flag */

  if (x==0.0f)
  ```

  If you are running an analysis through the user interface, you can enter this option in the **Other** field, under the **Advanced Settings** node on the **Configuration** pane. See `Other`.

  At the command line, add the flag to your analysis command.

```
polyspace-bug-finder -sources filename ^
-checkers BAD_FLOAT_OP -detect-bad-float-op-on-zero
```

### Risk

Checking for equality or inequality of two floating-point values might return unexpected results because floating-point representations are inexact and involve rounding errors.

### Fix

Instead of checking for equality of floating-point values:

```
if (val1 == val2)
```

check if their difference is less than a predefined tolerance value (for instance, the value `FLT_EPSILON` defined in `float.h`):

```
#include <float.h>
if(fabs(val1-val2) < FLT_EPSILON)
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Floats Inequality in `for`-loop

```
#include <stdio.h>
#include <math.h>
#include <float.h>

void func(void)
{
    float f;
    for (f = 1.0; f != 2.0; f = f + 0.1)
        (void)printf("Value: %f\n", f);
}
```

In this function, the `for`-loop tests the inequality of `f` and the number 2.0 as a stopping mechanism. The number of iterations is difficult to determine, or might be infinite, because of the imprecision in floating-point representation.

#### Correction — Change the Operator

One possible correction is to use a different operator that is not as strict. For example, an inequality like >= or <=.

```
#include <stdio.h>
#include <math.h>
```

```
#include <float.h>

void func(void)
{
    float f;
    for (f = 1.0; f <= 2.0; f = f + 0.1)
        (void)printf("Value: %f\n", f);
}
```

### Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `BAD_FLOAT_OP`
**Impact:** Medium

## Version History

**Introduced in R2013b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Format string specifiers and arguments mismatch

Format specifiers in `printf`-like functions do not match corresponding arguments

## Description

This defect occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

### Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

### Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
const char *message = "License not available";
int err_number = -4;
printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

In cases where integer promotion modifies the perceived data type of an argument, the analysis result shows both the original type and the type after promotion. The format specifier has to match the type after integer promotion.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Examples

### Printing a Float

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

#### Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

#### Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

### Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `STRING_FORMAT`
**Impact:** Low

## Version History

**Introduced in R2013b**

### See Also

`Find defects (-checkers)` | `Invalid use of standard library string routine`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

#### External Websites

Standard library output functions

# Function called from signal handler not asynchronous-safe

Call to interrupted function causes undefined program behavior

## Description

This defect occurs when a signal handler calls a function that is not asynchronous-safe according to the POSIX standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

### Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

### Fix

The POSIX standard defines these functions as asynchronous-safe. You can call these functions from a signal handler.

`_exit()`

`_Exit()`

`abort()`

`accept()`

`access()`

`aio_error()`

`aio_return()`

`aio_suspend()`

`alarm()`

`bind()`

`cfgetispeed()`

`cfgetospeed()`

`cfsetispeed()`

`cfsetospeed()`

`chdir()`

`getpgrp()`

`getpid()`

`getppid()`

`getsockname()`

`getsockopt()`

`getuid()`

`kill()`

`link()`

`linkat()`

`listen()`

`lseek()`

`lstat()`

`mkdir()`

`mkdirat()`

`mkfifo()`

`setsockopt()`

`setuid()`

`shutdown()`

`sigaction()`

`sigaddset()`

`sigdelset()`

`sigemptyset()`

`sigfillset()`

`sigismember()`

`signal()`

`sigpause()`

`sigpending()`

`sigprocmask()`

`sigqueue()`

`sigset()`

| | | |
|---|---|---|
| chmod() | mkfifoat() | sigsuspend() |
| chown() | mknod() | sleep() |
| clock_gettime() | mknodat() | sockatmark() |
| close() | open() | socket() |
| connect() | openat() | socketpair() |
| creat() | pathconf() | stat() |
| dup() | pause() | symlink() |
| dup2() | pipe() | symlinkat() |
| execl() | poll() | sysconf() |
| execle() | posix_trace_event() | tcdrain() |
| execv() | pselect() | tcflow() |
| execve() | pthread_kill() | tcflush() |
| faccessat() | pthread_self() | tcgetattr() |
| fchdir() | pthread_sigmask() | tcgetpgrp() |
| fchmod() | quick_exit() | tcsendbreak() |
| fchmodat() | raise() | tcsetattr() |
| fchown() | read() | tcsetpgrp() |
| fchownat() | readlink() | time() |
| fcntl() | readlinkat() | timer_getoverrun() |
| fdatasync() | recv() | timer_gettime() |
| fexecve() | recvfrom() | timer_settime() |
| fork() | recvmsg() | times() |
| fpathconf() | rename() | umask() |
| fstat() | renameat() | uname() |
| fstatat() | rmdir() | unlink() |
| fsync() | select() | unlinkat() |
| ftruncate() | sem_post() | utime() |
| futimens() | send() | utimensat() |
| getegid() | sendmsg() | utimes() |
| geteuid() | sendto() | wait() |
| getgid() | setgid() | waitpid() |
| getgroups() | setpgid() | write() |
| getpeername() | setsid() | |

Functions not in the previous table are not asynchronous-safe, and should not be called from a signal hander.

## Examples

### Call to `printf()` Inside Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler(int signum)
{
    /* Call function printf() that is not
    asynchronous-safe */
    printf("signal %d received.", signum);
    e_flag = 1;
}

int main(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, sizeof(char));
    if (info == NULL)
    {
        /* Handle Error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

In this example, sig_handler calls printf() when catching a signal. If the handler catches another signal while printf() is executing, the behavior of the program is undefined.

**Correction — Set Flag Only in Signal Handler**

Use your signal handler to set only the value of a flag. e_flag is of type volatile sig_atomic_t. sig_handler can safely access it asynchronously.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler1(int signum)
{
    int s0 = signum;
    e_flag = 1;
}

int func(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler1) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, 1);
    if (info == NULL)
    {
        /* Handle error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SIG_HANDLER_ASYNC_UNSAFE`
**Impact:** Medium

# Version History

**Introduced in R2017b**

## See Also

`Function called from signal handler not asynchronous-safe (strict)` | `Return from computational exception signal handler` | `Shared data access within signal handler` | `Signal call from within signal handler` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Function called from signal handler not asynchronous-safe (strict ISO C)

Call to interrupted function causes undefined program behavior

## Description

This defect occurs when a signal handler calls a function that is not asynchronous-safe according to the C standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

The C standard defines a stricter subset of functions as asynchronous-safe compared to the set of functions that are asynchronous-safe according to the POSIX standard. **Function called from signal handler not asynchronous-safe (strict ISO C)** reports a defect when a signal handler calls any function that is not part of that subset, even if the function is asynchronous-safe according to the POSIX standard.

To check for calls to functions that are not asynchronous-safe according to the POSIX standard, enable checker **Function called from signal handler not asynchronous-safe**.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

### Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

### Fix

The C standard defines the following functions as asynchronous-safe. You can call these functions from a signal handler:

- `abort()`
- `_Exit()`
- `quick_exit()`
- `signal()`

## Examples

### Call to `raise()` Inside Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}

void sig_handler(int signum)
{
    int s0 = signum;
    /* Call raise() */
    if (raise(SIGTERM) != 0) {
        /* Handle error */
    }
}

int finc(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` calls `raise()` when catching a signal. If the handler catches another signal while `raise()` is executing, the behavior of the program is undefined.

**Correction — Remove Call to `raise()` in Signal Handler**

According to the C standard, the only functions that you can safely call from a signal handler are `abort()`, `_Exit()`, `quick_exit()`, and `signal()`.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
```

```
{
    int s0 = signum;
    /* SIGTERM specific handling */
}
void sig_handler(int signum)
{
    int s0 = signum;


}

int func(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SIG_HANDLER_ASYNC_UNSAFE_STRICT`
**Impact:** Medium

# Version History

**Introduced in R2017b**

## See Also

`Function called from signal handler not asynchronous-safe` | `Shared data access within signal handler` | `Signal call from within signal handler` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Improper array initialization

Incorrect array initialization when using initializers

## Description

This defect occurs when Polyspace Bug Finder considers that an array initialization using initializers is incorrect.

This defect applies to normal and designated initializers. In C99, with designated initializers, you can place the elements of an array initializer in any order and implicitly initialize some array elements. The designated initializers use the array index to establish correspondence between an array element and an array initializer element. For instance, the statement `int arr[6] = { [4] = 29, [2] = 15 }` is equivalent to `int arr[6] = { 0, 0, 15, 0, 29, 0 }`.

You can use initializers incorrectly in one of the following ways.

| Issue | Risk | Possible Fix |
|---|---|---|
| In your initializer for a one-dimensional array, you have more elements than the array size. | Unused array initializer elements indicate a possible coding error. | Increase the array size or remove excess elements. |
| You place the braces enclosing initializer values incorrectly. | Because of the incorrect placement of braces, some array initializer elements are not used.<br>Unused array initializer elements indicate a possible coding error. | Place braces correctly. |
| In your designated initializer, you do not initialize the first element of the array explicitly. | The implicit initialization of the first array element indicates a possible coding error. You possibly overlooked the fact that array indexing starts from 0. | Initialize all elements explicitly. |
| In your designated initializer, you initialize an element twice. | The first initialization is overridden.<br>The redundant first initialization indicates a possible coding error. | Remove the redundant initialization. |
| You use designated and nondesignated initializers in the same initialization. | You or another reviewer of your code cannot determine the size of the array by inspection. | Use either designated or nondesignated initializers. |

#### Fix

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Incorrectly Placed Braces (C Only)

```
int arr[2][3]
= {{1, 2},
    {3, 4},
    {5, 6}
};
```

In this example, the array `arr` is initialized as `{1,2,0,3,4,0}`. Because the initializer contains `{5,6}`, you might expect the array to be initialized `{1,2,3,4,5,6}`.

#### Correction — Place Braces Correctly

One possible correction is to place the braces correctly so that all elements are explicitly initialized.

```
int a1[2][3]
= {{1, 2, 3},
    {4, 5, 6}
};
```

### First Element Not Explicitly Initialized

```
int arr[5]
= {
    [1] = 2,
    [2] = 3,
    [3] = 4,
    [4] = 5
};
```

In this example, `arr[0]` is not explicitly initialized. It is possible that the programmer did not consider that the array indexing starts from 0.

#### Correction — Explicitly Initialize All Elements

One possible correction is to initialize all elements explicitly.

```
int arr[5]
= {
    [0] = 1,
    [1] = 2,
    [2] = 3,
    [3] = 4,
    [4] = 5
};
```

### Element Initialized Twice

```
int arr[5]
= {
    [0] = 1,
    [1] = 2,
    [2] = 3,
    [2] = 4,
    [4] = 5
};
```

In this example, arr[2] is initialized twice. The first initialization is overridden. In this case, because arr[3] was not explicitly initialized, it is possible that the programmer intended to initialize arr[3] when arr[2] was initialized a second time.

#### Correction — Fix Redundant Initialization

One possible correction is to eliminate the redundant initialization.

```
int arr[5]
= {
    [0] = 1,
    [1] = 2,
    [2] = 3,
    [3] = 4,
    [4] = 5
};
```

### Mix of Designated and Nondesignated Initializers

```
int arr[]
= {
    [0] = 1,
    [3] = 3,
    4,
    [5] = 5,
    6
    };
```

In this example, because a mix of designated and nondesignated initializers are used, it is difficult to determine the size of arr by inspection.

#### Correction — Use Only Designated Initializers

One possible correction is to use only designated initializers for array initialization.

```
int arr[]
= {
    [0] = 1,
    [3] = 3,
    [4] = 4,
    [5] = 5,
    [6] = 6
};
```

### Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `IMPROPER_ARRAY_INIT`
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Incorrect data type passed to va_arg

Data type of variadic function argument does not match type in va_arg call

## Description

This defect occurs when the data type in a va_arg call does not match the data type of the variadic function argument that va_arg reads.

For instance, you pass an unsigned char argument to a variadic function func. Because of default argument promotion, the argument is promoted to int. When you use a va_arg call that reads an unsigned char argument, a type mismatch occurs.

```
void func (int n, ...) {
   ...
   va_list args;
   va_arg(args, unsigned char);
   ...
}

void main(void) {
   unsigned char c;
   func(1,c);
}
```

### Risk

In a variadic function (function with variable number of arguments), you use va_arg to read each argument from the variable argument list (va_list). The va_arg use does not guarantee that there actually exists an argument to read or that the argument data type matches the data type in the va_arg call. You have to make sure that both conditions are true.

Reading an incorrect type with a va_arg call can result in undefined behavior. Because function arguments reside on the stack, you might access an unwanted area of the stack.

### Fix

Make sure that the data type of the argument passed to the variadic function matches the data type in the va_arg call.

Arguments of a variadic function undergo default argument promotions. The argument data types of a variadic function cannot be determined from a prototype. The arguments of such functions undergo default argument promotions (see Sec. 6.5.2.2 and 7.15.1.1 in the C99 Standard). Integer arguments undergo integer promotion and arguments of type float are promoted to double. For integer arguments, if a data type can be represented by an int, for instance, char or short, it is promoted to an int. Otherwise, it is promoted to an unsigned int. All other arguments do not undergo promotion.

To avoid undefined and implementation-defined behavior, minimize the use of variadic functions. Use the checkers for MISRA C:2012 Rule 17.1 or MISRA C++:2008 Rule 8-4-1 to detect use of variadic functions.

## Examples

### char Used as Function Argument Type and va_arg argument

```
#include <stdarg.h>
#include <stdio.h>

unsigned char func(size_t count, ...) {
    va_list ap;
    unsigned char result = 0;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, unsigned char);
    }
    va_end(ap);
    return result;
}

void func_caller(void) {
    unsigned char c = 0x12;
    (void)func(1, c);
}
```

In this example, func takes an unsigned char argument, which undergoes default argument promotion to int. The data type in the va_arg call is still unsigned char, which does not match the int argument type.

#### Correction — Use int as va_arg Argument

One possible correction is to read an int argument with va_arg.

```
#include <stdarg.h>
#include <stdio.h>

unsigned char func(size_t count, ...) {
    va_list ap;
    unsigned char result = 0;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
    }
    va_end(ap);
    return result;
}

void func_caller(void) {
    unsigned char c = 0x12;
    (void)func(1, c);
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** VA_ARG_INCORRECT_TYPE
**Impact:** Medium

## Version History

**Introduced in R2018a**

## See Also

`Invalid va_list argument` | `Too many va_arg calls for current argument list` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Incorrect pointer scaling

Implicit scaling in pointer arithmetic might be ignored

## Description

This defect occurs when Polyspace Bug Finder considers that you are ignoring the implicit scaling in pointer arithmetic.

For instance, the defect can occur in the following situations.

| Situation | Risk | Possible Fix |
|---|---|---|
| You use the `sizeof` operator in arithmetic operations on a pointer. | The `sizeof` operator returns the size of a data type in number of bytes.<br>Pointer arithmetic is already implicitly scaled by the size of the data type of the pointed variable. Therefore, the use of `sizeof` in pointer arithmetic produces unintended results. | Do not use `sizeof` operator in pointer arithmetic. |
| You perform arithmetic operations on a pointer, and then apply a cast. | Pointer arithmetic is implicitly scaled. If you do not consider this implicit scaling, casting the result of a pointer arithmetic produces unintended results. | Apply the cast before the pointer arithmetic. |

### Fix

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Examples

### Use of `sizeof` Operator

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;
    int value_in_position_2 = *(ptr + 2*(sizeof(int)));
}
```

In this example, the operation `2*(sizeof(int))` returns twice the size of an `int` variable in bytes. However, because pointer arithmetic is implicitly scaled, the number of bytes by which `ptr` is offset is `2*(sizeof(int))*(sizeof(int))`.

In this example, the incorrect scaling shifts `ptr` outside the bounds of the array. Therefore, a **Pointer access out of bounds** error appears on the * operation.

#### Correction — Remove `sizeof` Operator

One possible correction is to remove the `sizeof` operator.

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2);
}
```

### Cast Following Pointer Arithmetic

```
int func(void) {
    int x = 0;
    char r = *(char *)(&x + 1);
    return r;
}
```

In this example, the operation `&x + 1` offsets `&x` by `sizeof(int)`. Following the operation, the resulting pointer points outside the allowed buffer. When you dereference the pointer, a **Pointer access out of bounds** error appears on the * operation.

#### Correction — Apply Cast Before Pointer Arithmetic

If you want to access the second byte of `x`, first cast `&x` to a `char*` pointer and then perform the pointer arithmetic. The resulting pointer is offset by `sizeof(char)` bytes and still points within the allowed buffer, whose size is `sizeof(int)` bytes.

```
int func(void) {
    int x = 0;
    char r = *((char *)(&x )+ 1);
    return r;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `BAD_PTR_SCALING`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Incorrect type data passed to va_start

Data type of second argument to `va_start` macro leads to undefined behavior

## Description

This defect occurs when the second argument of the `va_start` macro has one of these data types:

- A data type that changes when undergoing default argument promotion.

  For instance, `char` and `short` undergo promotion to `int` or `unsigned int` and `float` undergoes promotion to `double`. The types `int` and `double` do not change under default argument promotion.
- (C only) A register type or a data type declared with the `register` qualifier.
- (C++ only) A reference data type.
- (C++ only) A data type that has a nontrivial copy constructor or a nontrivial move constructor.

### Risk

In a variadic function or function with variable number of arguments:

```
void multipleArgumentFunction(int someArg, short rightmostFixedArg, ...) {
    va_list myList;
    va_start(myList, rightmostFixedArg);
    ...
    va_end(myList);
}
```

The `va_start` macro initializes a variable argument list so that additional arguments to the variadic function after the fixed parameters can be captured in the list. According to the C11 and C++14 Standards, if you use one of the flagged data types for the second argument of the `va_start` macro (for instance, `rightmostFixedArg` in the preceding example), the behavior is undefined.

If the data type involves a nontrivial copy constructor, the behavior is implementation-defined. For instance, whether the copy constructor is invoked in the call to `va_start` depends on the compiler.

### Fix

When using the `va_start` macro, try to use the types `int`, `unsigned int` or `double` for the rightmost named parameter of the variadic function. Then, use this parameter as the second argument of the `va_start` macro.

For instance, in this example, the rightmost named parameter of the variadic function has a supported data type `int`:

```
void multipleArgumentFunction(int someArg, int rightmostFixedArg, ...) {
    va_list myList;
    va_start(myList, rightmostFixedArg);
    ...
    va_end(myList);
}
```

To avoid undefined and implementation-defined behavior, minimize the use of variadic functions. Use the checkers for MISRA C:2012 Rule 17.1 or MISRA C++:2008 Rule 8-4-1 to detect use of variadic functions.

## Examples

### Incorrect Data Types for Second Argument of va_start

```
#include <string>
#include <cstdarg>

double addVariableNumberOfDoubles(double* weight, short num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}

double addVariableNumberOfFloats(float* weight, int num, std::string s, ...) {
    float sum=0.0;
    va_list list;
    va_start(list, s);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, float);
    }
    va_end(list);
    return sum;
}
```

In this example, the checker flags the call to va_start in:

- addVariableNumberOfDoubles because the argument has type short, which undergoes default argument promotion to int.
- addVariableNumberOfFloats because the argument has type std::string, which has a nontrivial copy constructor.

#### Correction — Fix Data Type for Second Argument of va_start

Make sure that the second argument of the va_start macro has a supported data type. In the following corrected example:

- In addVariableNumberOfDoubles, the data type of the last named parameter of the variadic function is changed to int.
- In addVariableNumberOfFloats, the second and third parameters of the variadic function are switched so that data type of the last named parameter is int.

```
#include <string>
#include <cstdarg>

double addVariableNumberOfDoubles(double* weight, int num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}

double addVariableNumberOfFloats(double* weight, std::string s, int num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `VA_START_INCORRECT_TYPE`
**Impact:** Medium

# Version History

**Introduced in R2019a**

## See Also

`Incorrect data type passed to va_arg` | `Too many va_arg calls for current argument list` | `Incorrect use of va_start` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Incorrect use of offsetof in C++

Incorrect arguments to `offsetof` macro causes undefined behavior

## Description

This defect occurs when you pass arguments to the `offsetof` macro for which the behavior of the macro is not defined.

The `offsetof` macro:

```
offsetof(classType, aMember)
```

returns the offset in bytes of the data member `aMember` from the beginning of an object of type `classType`. For use in `offsetof`, `classType` and `aMember` have certain restrictions:

- `classType` must be a standard layout class.

  For instance, it must not have `virtual` member functions. For more information on the requirements for a standard layout class, see C++ named requirements: StandardLayoutType.
- `aMember` must not be static.
- `aMember` must not be a member function.

The checker flags uses of the `offsetof` macro where the arguments violate one or more of these restrictions.

### Risk

Violating the restrictions on the arguments of the `offsetof` macro leads to undefined behavior.

### Fix

Use the `offsetof` macro only on nonstatic data members of a standard layout class.

The result details state which restriction on the `offsetof` macro is violated. Fix the violation.

## Examples

### Use of `offsetof` Macro with Nonstandard Layout Class

```
#include <cstddef>

class myClass {
     int member1;
  public:
     int member2;
};

void func() {
  size_t off = offsetof(myClass, member2);
  // ...
}
```

In this example, the class `myClass` has two data members with different access control, one private and the other public. Therefore, the class does not satisfy the requirements of a standard layout class and cannot be used with the `offsetof` macro.

#### Correction — Use Uniform Access Control for All Data Members

If the use of `offsetof` is important for the application, make sure that the first argument is a class with a standard layout. For instance, see if you can work around the need for a public data member.

```
#include <cstddef>

class myClass {
     int member1;
     int member2;
  public:
     int getMember2(void) { return member2;}
     friend void func(void);
};

void func() {
  size_t off = offsetof(myClass, member2);
  // ...
}
```

## Result Information

**Group:** Programming
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `OFFSETOF_MISUSE`
**Impact:** Medium

# Version History

**Introduced in R2019a**

## See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Incorrect use of va_start

`va_start` is called in a non-variadic function or called with a second argument that is not the rightmost parameter of a variadic function

## Description

This defect occurs when you use the `va_start` macro in a way that violates its specifications.

In a variadic function or function with variable number of arguments:

```
void multipleArgumentFunction(int someArg, int rightmostFixedArg, ...) {
    va_list myList;
    va_start(myList, rightmostFixedArg);
    ...
    va_end(myList);
}
```

The `va_start` macro initializes a variable argument list so that additional arguments to the variadic function after the fixed parameters can be captured in the list. In the preceding example, the `va_start` macro initializes `myList` so that it can capture arguments after `rightmostFixedArg`.

You can violate the specifications of `va_start` in multiple ways. For instance:

- You call `va_start` in a non-variadic function.
- The second argument of `va_start` is not the rightmost fixed parameter of the variadic function.

### Risk

Violating the specifications of the `va_start` macro can result in compilation errors. If the compiler fails to detect the violation, the violation can result in undefined behavior.

### Fix

Make sure that:

- The `va_start` macro is used in a variadic function
- The second argument of the `va_start` macro is the rightmost fixed parameter of the variadic function.

To avoid undefined and implementation-defined behavior, minimize the use of variadic functions. Use the checkers for `MISRA C:2012 Rule 17.1` or `MISRA C++:2008 Rule 8-4-1` to detect use of variadic functions.

## Examples

### Incorrect Argument to va_start

```
#include <stdarg.h>

double addVariableNumberOfDoubles(int num, double* weight, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}
```

In this example, the rightmost fixed parameter to the `addVariableNumberOfDoubles` function is `weight`. However, a different parameter is used as the second argument to the `va_start` macro.

**Correction — Switch Order of Fixed Parameters of Variadic Function**

One possible correction is to modify the order of fixed parameters to the variadic function so that the rightmost fixed parameter is used for the `va_start` macro.

```
#include <stdarg.h>

double addVariableNumberOfDoubles(double* weight, int num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `VA_START_MISUSE`
**Impact:** Medium

# Version History

**Introduced in R2019a**

## See Also

`Incorrect data type passed to va_arg` | `Too many va_arg calls for current argument list` | `Incorrect type data passed to va_start` | `Find defects (-checkers)`

**Topics**

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Incorrect value forwarding

Forwarded object might be modified unexpectedly

## Description

This defect occurs when:

- You use `std::move` to forward a forwarding reference to a function, including objects of type `auto&&` or `T&&`.
- You use `std::forward` to forward an `rvalue` reference to a function.

Consider the following code. In the function template `callMethod`, the parameter `param1` is a forwarding reference and the parameter `param2` is an `rvalue` reference. When forwarding these parameters to the function `foo()`, the code incorrectly uses `std::move` on the forwarding reference `param1` and `std::forward` on the `rvalue` reference `param2`.

```
#include<string>
#include<iostream>

template <class T> class  bar{};

template<typename T>
void foo(std::string&& s, bar<T>&&){
    //...
}

template <typename T>
void callMethod(T&& param1, bar<T>&& param2) {

    foo(std::move(param1),std::forward<bar<T>&&>(param2));
}
```

Forwarding references are declared as `T&&` or `auto&&`, and `rvalue` references are declared as `&&`. Confusing forwarding references with `rvalue` references results in incorrect value forwarding.

Polyspace does not report this defect if `std::move` or `std::forward` does not forward a reference to a function. For instance, this code does not forward the `rvalue` reference `b1` and the forwarding reference `b2` to another function. Using `std::move` with `b2` or using `std::forward` with `b1` does not cause a defect.

```
template <typename T1, typename T2>
void func(T1& b1, T2&& b2)
{
    const T1& b10 = std::forward<B>(b1);
    const T2& b20 = std::forward<B>(b2);
    const T1& b11 = std::move(b1);
    const T2& b21 = std::move(b2);
}
```

### Risk

Using `std::move` with forwarding references can result in an unexpected modification of an lvalue. Using `std::forward` with `rvalue` references is error-prone and can increase the complexity of your code.

### Fix

- If you forward an `rvalue` reference to a function, use `std::move` to cast the reference to an `rvalue`.
- If you forward a forwarding or universal reference to a function, use `std::forward` to cast the reference to an `rvalue` only if the object is bound to an `rvalue`.

## Examples

### Values Forwarded Incorrectly

```
#include <cstdint>
#include <string>
#include <utility>


class A
{
public:
    explicit A(std::string&& s)
        : str(std::move(s))
    {
    }

private:
    std::string str;
};

template <typename ...T>
void f1(T...t);



template <typename T1, typename T2>
void func(T1&& t1, T2& t2)
{
    f1(std::move(t1));
    f1(std::forward<T1>(t1));//Suggested fix

    f1(std::forward<T2>(t2));
    f1(std::move(t2));//Suggested fix
}

void func_auto(A& var)
{
    auto&& var1 = var;
    f1(std::move(var1));
    f1(std::forward<decltype(var1)>(var1));//Suggested fix
}
```

```
void main()
{
    int32_t i;
    func(0, i);
}
```

In this example, the template function `func` forwards parameters `t1` and `t2` to function `f1`.

- Because the `rvalue` reference `t2` is forwarded by calling `std::forward`, Polyspace reports a defect.
- Because the forwarding reference `t1` is forwarded by calling `std::move`, Polyspace reports a defect. If `t1` is initialized with an lvalue, the move might result in an unexpected modification of the parameter.
- Similarly, Polyspace reports a defect on the use of `std::move` in `func_auto` because objects of type `auto`&& are considered as forwarding references.

For each defect, the subsequent lines of code shows the suggested fix.

## Result Information

**Group:** Programming
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `INCORRECT_VALUE_FORWARDING`
**Impact:** High

# Version History

**Introduced in R2020b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Inline constraint not respected

Modifiable static variable is modified in nonstatic inline function

## Description

This defect occurs when you refer to a file scope modifiable static variable or define a local modifiable static variable in a nonstatic inlined function. The checker considers a variable as modifiable if it is not `const`-qualified.

For instance, `var` is a modifiable `static` variable defined in an `inline` function `func`. `g_step` is a file scope modifiable static variable referred to in the same inlined function.

```
static int g_step;
inline void func (void) {
   static int var = 0;
   var += g_step;
}
```

### Risk

When you modify a static variable in multiple function calls, you expect to modify the same variable in each call. For instance, each time you call `func`, the same instance of `var1` is incremented but a separate instance of `var2` is incremented.

```
void func(void) {
   static var1 = 0;
   var2 = 0;
   var1++;
   var2++;
}
```

If a function has an inlined and non-inlined definition (in separate files), when you call the function, the C standard allows compilers to use either the inlined or the non-inlined form (see ISO/IEC 9899:2011, sec. 6.7.4). If your compiler uses an inlined definition in one call and the non-inlined definition in another, you are no longer modifying the same variable in both calls. This behavior defies the expectations from a static variable.

### Fix

Use one of these fixes:

- If you do not intend to modify the variable, declare it as `const`.

  If you do not modify the variable, there is no question of unexpected modification.
- Make the variable non-`static`. Remove the `static` qualifier from the declaration.

  If the variable is defined in the function, it becomes a regular local variable. If defined at file scope, it becomes an extern variable. Make sure that this change in behavior is what you intend.
- Make the function `static`. Add a `static` qualifier to the function definition.

If you make the function `static`, the file with the inlined definition always uses the inlined definition when the function is called. Other files use another definition of the function. The question of which function definition gets used is not left to the compiler.

## Examples

### Static Variable Use in Inlined and External Definition

```
/* file1. c  : contains inline definition of get_random()*/

inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

In this example, `get_random()` has an inline definition in `file1.c` and an external definition in `file2.c`. When `get_random` is called in `file1.c`, compilers are free to choose whether to use the inline or the external definition.

Depending on the definition used, you might or might not modify the version of `m_z` and `m_w` in the inlined version of `get_random()`. This behavior contradicts the usual expectations from a static variable. When you call `get_random()`, you expect to always modify the same `m_z` and `m_w`.

### Correction — Make Inlined Function Static

One possible correction is to make the inlined `get_random()` static. Irrespective of your compiler, calls to `get_random()` in `file1.c` then use the inlined definition. Calls to `get_random()` in other files use the external definition. This fix removes the ambiguity about which definition is used and whether the static variables in that definition are modified.

```
/* file1. c  : contains inline definition of get_random()*/

static inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `INLINE_CONSTRAINT_NOT_RESPECTED`
**Impact:** Medium

## Version History

**Introduced in R2018a**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Invalid assumptions about memory organization

Address is computed by adding or subtracting from address of a variable

## Description

This defect occurs when you compute the address of a variable in the stack by adding or subtracting from the address of another non-array variable.

### Risk

When you compute the address of a variable in the stack by adding or subtracting from the address of another variable, you assume a certain memory organization. If your assumption is incorrect, accessing the computed address can be invalid.

### Fix

Do not perform an access that relies on assumptions about memory organization.

## Examples

### Reliance on Memory Organization

```
void func(void) {
    int var1 = 0x00000011, var2;
    *(&var1 + 1) = 0;
}
```

In this example, the programmer relies on the assumption that `&var1 + 1` provides the address of `var2`. Therefore, an **Invalid assumptions about memory organization** appears on the `+` operation. In addition, a **Pointer access out of bounds** error also appears on the dereference.

#### Correction — Do Not Rely on Memory Organization

One possible correction is not perform direct computation on addresses to access separately declared variables.

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `INVALID_MEMORY_ASSUMPTION`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

**Topics**

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Invalid file position

`fsetpos()` is invoked with a file position argument not obtained from `fgetpos()`

## Description

This defect occurs when the file position argument of `fsetpos()` uses a value that is not obtained from `fgetpos()`.

### Risk

The function `fgetpos(FILE *stream, fpos_t *pos)` gets the current file position of the stream. When you use any other value as the file position argument of `fsetpos(FILE *stream, const fpos_t *pos)`, you might access an unintended location in the stream.

### Fix

Use the value returned from a successful call to `fgetpos()` as the file position argument of `fsetpos()`.

## Examples

### `memset()` Sets File Position Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset' */
    (void)memset(&offset, 0, sizeof(offset));

    /* Read data from file */

    /* Return to the initial position. offset was not
    returned from a call to fgetpos()    */
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

In this example, `fsetpos()` uses `offset` as its file position argument. However, the value of `offset` is set by `memset()`. The preceding code might access the wrong location in the stream.

### Correction — Use a File Position Returned From `fgetpos()`

Call `fgetpos()`, and if it returns successfully, use the position argument in your call to `fsetpos()`.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset'
    using fgetpos() */
    if (fgetpos(file, &offset) != 0)
    {
        /* Handle error */
    }

    /* Read data from file */

    /* Back to the initial position */
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `INVALID_FILE_POS`
**Impact:** Medium

# Version History

**Introduced in R2017b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Invalid use of = operator

Assignment in conditional statement

## Description

This defect occurs when an assignment is made inside the predicate of a conditional, such as `if` or `while`.

In C and C++, a single equal sign is an assignment not a comparison. Using a single equal sign in a conditional statement can indicate a typo or a mistake.

### Risk

- Conditional statement tests the wrong values— The single equal sign operation assigns the value of the right operand to the left operand. Then, because this assignment is inside the predicate of a conditional, the program checks whether the new value of the left operand is nonzero or not NULL.
- Maintenance and readability issues — Even if the assignment is intended, someone reading or updating the code can misinterpret the assignment as an equality comparison instead of an assignment.

### Fix

- If the assignment is a bug, to check for equality, add a second equal sign (==).
- If the assignment inside the conditional statement was intentional, to improve readability, separate the assignment and the test. Move the assignment outside the control statement. In the control statement, simply test the result of the assignment.

  If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

  - “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
  - “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
  - “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Examples

### Single Equal Sign Inside an `if` Condition

```
#include <stdio.h>

void bad_equals_ex(int alpha, int beta)
{
    if(alpha = beta)
    {
        printf("Equal\n");
    }
}
```

The equal sign is flagged as a defect because the assignment operator is used within the predicate of the if-statement. The predicate assigns the value `beta` to `alpha`, then implicitly tests whether `alpha` is true or false.

#### Correction — Change Expression to Comparison

One possible correction is adding an additional equal sign. This correction changes the assignment to a comparison. The if condition compares whether `alpha` and `beta` are equal.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha == beta)
    {
        printf("Equal\n");
    }
}
```

#### Correction — Assignment and Comparison Inside the `if` Condition

If an assignment must be made inside the predicate, a possible correction is adding an explicit comparison. This correction assigns the value of `beta` to `alpha`, then explicitly checks whether `alpha` is nonzero. The code is clearer.

```
#include <stdio.h>

int assignment_not_zero(int alpha, int beta)
{
    if((alpha = beta) != 0)
    {
        return alpha;
    }
    else
    {
        return 0;
    }
}
```

#### Correction — Move Assignment Outside the `if` Statement

If the assignment can be made outside the control statement, one possible correction is to separate the assignment and comparison. This correction assigns the value of `beta` to `alpha` before the if. Inside the if-condition, only `alpha` is given to test if `alpha` is nonzero or not NULL.

```
#include <stdio.h>

void assign_and_print(int alpha, int beta)
{
    alpha = beta;
    if(alpha)
    {
        printf("%d", alpha);
    }
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `BAD_EQUAL_USE`
**Impact:** Medium

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Invalid use of == (equality) operator`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Invalid use of == operator

Equality operation in assignment statement

## Description

This defect occurs when you use an equality operator instead of an assignment operator in a simple statement.

### Risk

The use of == operator instead of an = operator can silently produce incorrect results. If you intended to assign a value to a variable, the assignment does not occur. The variable retains its previous value or if not initialized previously, stays uninitialized.

### Fix

Use the = (assignment) operator instead of the == (equality) operator.

The check appears on chained assignment and equality operators such as:

```
compFlag = val1 == val2;
```

For better readability of your code, place the equality check in parenthesis.

```
compFlag = (val1 == val2);
```

If the use of == operator is intended, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Equality Evaluation in for-Loop

```
void populate_array(void)
{
    int i = 0;
    int j = 0;
    int array[4];

    for (j == 5; j < 9; j++) {
        array[i] = j;
        i++;
    }
}
```

Inside the `for`-loop, the statement `j == 5` tests whether `j` is equal to 5 instead of setting `j` to 5. The `for`-loop iterates from 0 to 8 because `j` starts with a value of 0, not 5. A by-product of the invalid equality operator is an out-of-bounds array access in the next line.

#### Correction — Change to Assignment Operator

One possible correction is to change the == operator to a single equal sign (=). Changing the == sign resolves both defects because the `for`-loop iterates the intended number of times.

```
void populate_array(void)
{
    int i = 0;
    int j = 0;
    int array[4];

    for (j = 5; j < 9; j++) {
        array[i] = j;
        i++;
    }
}
```

### Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `BAD_EQUAL_EQUAL_USE`
**Impact:** High

## Version History

**Introduced in R2013b**

### See Also

`Find defects (-checkers)` | `Invalid use of = (assignment) operator`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Invalid use of standard library routine

Wrong arguments to standard library function

## Description

This defect occurs when you use invalid arguments with a function from the standard library. This defect picks up errors related to other functions not covered by float, integer, memory, or string standard library routines.

### Risk

Invalid arguments to a standard library function result in undefined behavior.

### Fix

The fix depends on the root cause of the defect. For instance, the argument to a `printf` function can be `NULL` because a pointer was initialized with `NULL` and the initialization value was not overwritten along a specific execution path.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Calling `printf` Without a String

```
#include <stdio.h>
#include <stdlib.h>

void print_null(void) {

  printf(NULL);
}
```

The function `printf` takes only string input arguments or format specifiers. In this function, the input value is NULL, which is not a valid string.

#### Correction — Use Compatible Input Arguments

One possible correction is to change the input arguments to fit the requirements of the standard library routine. In this example, the input argument was changed to a character.

```
#include <stdio.h>
```

```
void print_null(void) {
    char zero_val = '0';
    printf((const char*)zero_val);
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `OTHER_STD_LIB`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Invalid use of standard library integer routine` | `Invalid use of standard library floating point routine` | `Invalid use of standard library memory routine` | `Invalid use of standard library string routine`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Invalid va_list argument

Variable argument list used after invalidation with `va_end` or not initialized with `va_start` or `va_copy`

## Description

This defect occurs when you use a `va_list` variable as an argument to a function in the `vprintf` group but:

- You do not initialize the variable previously using `va_start` or `va_copy`.
- You invalidate the variable previously using `va_end` and do not reinitialize it.

For instance, you call the function `vsprintf` as `vsprintf (buffer,format, args)`. However, before the function call, you do not initialize the `va_list` variable `args` using either of the following:

- `va_start(args, paramName)`. `paramName` is the last named argument of a variable-argument function. For instance, for the function definition `void func(int n, char c, ...) {}`, `c` is the last named argument.
- `va_copy(args, anotherList)`. `anotherList` is another valid `va_list` variable.

### Risk

The behavior of an uninitialized `va_list` argument is undefined. Calling a function with an uninitialized `va_list` argument can cause stack overflows.

### Fix

Before using a `va_list` variable as function argument, initialize it with `va_start` or `va_copy`.

Clean up the variable using `va_end` only after all uses of the variable.

## Examples

### `va_list` Variable Used Following Call to va_end

```
#include <stdarg.h>
#include <stdio.h>

int call_vfprintf(int line, const char *format, ...) {
    va_list ap;
    int r=0;

    va_start(ap, format);
    r = vfprintf(stderr, format, ap);
    va_end(ap);

    r += vfprintf(stderr, format, ap);
    return r;
}
```

In this example, the `va_list` variable `ap` is used in the `vfprintf` function, after the `va_end` macro is called.

#### Correction — Call va_end After Using va_list Variable

One possible correction is to call `va_end` only after all uses of the `va_list` variable.

```
#include <stdarg.h>
#include <stdio.h>

int call_vfprintf(int line, const char *format, ...) {
    va_list ap;
    int r=0;

    va_start(ap, format);
    r = vfprintf(stderr, format, ap);
    r += vfprintf(stderr, format, ap);
    va_end(ap);

    return r;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `INVALID_VA_LIST_ARG`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)` | `Incorrect data type passed to va_arg`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Memory comparison of float-point values

Object representation of floating-point values can be different (same) for equal (not equal) floating-point values

## Description

This defect occurs when you compare the object representation of floating-point values or the object representation of structures containing floating-point members. When you use the functions memcmp, bcmp, or wmemcmp to perform the bit pattern comparison, the defect is raised.

### Risk

The object representation of floating-point values uses specific bit patterns to encode those values. Floating-point values that are equal, for instance -0.0 and 0.0 in the IEC 60559 standard, can have different bit patterns in their object representation. Similarly, floating-point values that are not equal can have the same bit pattern in their object representation.

### Fix

When you compare structures containing floating-point members, compare the structure members individually.

To compare two floating-point values, use the == or != operators. If you follow a standard that discourages the use of these operators, such as MISRA, ensure that the difference between the floating-point values is within an acceptable range.

## Examples

### Using memcmp to Compare Structures with Floating-Point Members

```
#include <string.h>

typedef struct {
    int i;
    float f;
} myStruct;

extern void initialize_Struct(myStruct *);

int func_cmp(myStruct *s1, myStruct *s2) {
/* Comparison between structures containing
* floating-point members */
    return memcmp
        ((const void *)s1, (const void *)s2, sizeof(myStruct));
}

void func(void) {
    myStruct s1, s2;
    initialize_Struct(&s1);
    initialize_Struct(&s2);
    (void)func_cmp(&s1, &s2);
}
```

In this example, `func_cmp()` calls `memcmp()` to compare the object representations of structures `s1` and `s2`. The comparison might be inaccurate because the structures contain floating-point members.

#### Correction — Compare Structure Members Individually

One possible correction is to compare the structure members individually and to ensure that the difference between the floating-point values is within an acceptable range defined by `ESP`.

```
#include <string.h>
#include <math.h>

typedef struct {
    int i;
    float f;
} myStruct;

extern void initialize_Struct(myStruct *);

#define ESP 0.00001

int func_cmp(myStruct *s1, myStruct *s2) {

/*Structure members are compared individually */
    return ((s1->i == s2->i) &&
            (fabsf(s1->f - s2->f) <= ESP));
}

void func(void) {
    myStruct s1, s2;
    initialize_Struct(&s1);
    initialize_Struct(&s2);
    (void)func_cmp(&s1, &s2);
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `MEMCMP_FLOAT`
**Impact:** Low

# Version History

**Introduced in R2018a**

## See Also

`Find defects (-checkers)` | `Floating point comparison with equality operators` | `Memory comparison of padding data` | `Memory comparison of strings`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"

“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Memory comparison of padding data

memcmp compares data stored in structure padding

## Description

This defect occurs when you use the memcmp function to compare two structures as a whole. In the process, you compare meaningless data stored in the structure padding.

For instance:

```
struct structType {
    char member1;
    int member2;
    .
    .
};

structType var1;
structType var2;
.
.
if(memcmp(&var1,&var2,sizeof(var1)))
{...}
```

### Risk

If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. For an example of padding, see Higher Estimate of Size of Local Variables.

The content of these extra padding bytes is meaningless. The C Standard allows the content of these bytes to be indeterminate, giving different compilers latitude to implement their own padding. If you perform a byte-by-byte comparison of structures with memcmp, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.

### Fix

Instead of comparing two structures in one attempt, compare the structures member by member.

For efficient code, write a function that does the comparison member by member. Use this function for comparing two structures.

You can use memcmp for byte-by-byte comparison of structures only if you know that the structures do not contain padding. Typically, to prevent padding, you use specific attributes or pragmas such as #pragma pack. However, these attributes or pragmas are not supported by all compilers and make your code implementation-dependent. If your structures contain bit-fields, using these attributes or pragmas cannot prevent padding.

## Examples

### Structures Compared with memcmp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{

    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    if (0 == memcmp(left, right, sizeof(S_Padding)))
    {
        return 1;
    }
    else
        return 0;
}
```

In this example, `memcmp` compares byte-by-byte the two structures that `left` and `right` point to. Even if the values stored in the structure members are the same, the comparison can show an inequality if the meaningless values in the padding bytes are not the same.

#### Correction — Compare Structures Member by Member

One possible correction is to compare individual structure members.

**Note** You can compare entire arrays by using `memcmp`. All members of an array have the same data type. Padding bytes are not required to store arrays.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
```

```
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{
    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    return ((left->c == right->c) &&
            (left->i == right->i) &&
            (left->bf1 == right->bf1) &&
            (left->bf2 == right->bf2) &&
            (memcmp(left->buffer, right->buffer, 20) == 0));
}
```

### Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `MEMCMP_PADDING_DATA`
**Impact:** Medium

## Version History

**Introduced in R2017a**

### See Also

`Find defects (-checkers)` | `Memory comparison of strings`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Memory comparison of strings

memcmp compares data stored in strings after the null terminator

## Description

This defect occurs when:

- You compare two strings byte-by-byte with the memcmp function.
- The number of bytes compared is such that you compare meaningless data stored after the null terminator.

For instance:

```
memcmp(string1, string2, sizeof(string1))
```

can compare bytes in the string after the null terminator.

### Risk

The null terminator signifies the end of a string. Comparison of bytes after the null terminator is meaningless. You might reach the false conclusion that two strings are not equal, even if the bytes before the null terminator store the same value.

### Fix

Use strcmp for string comparison. The function compares strings only up to the null terminator.

If you use memcmp for a byte-by-byte comparison of two strings, avoid comparison of bytes after the null terminator. Determine the number of bytes to compare by using the strlen function.

## Examples

### Strings Compared with memcmp

```
#include <stdio.h>
#include <string.h>

#define SIZE20 20

int func()
{
    char s1[SIZE20] =  "abc";
    char s2[SIZE20] =  "abc";

    return memcmp(s1, s2, sizeof(s1));
}
```

In this example, sizeof returns the length of the entire array s1, which is 20. However, only the first three bytes of the string are relevant.

Even though s1 and s2 hold the same value, the comparison with memcmp can show a false inequality.

#### Correction — Use `strlen` to Determine Number of Bytes to Compare

One possible correction is to determine the number of bytes to compare using the `strlen` function. `strlen` returns the number of bytes *before* the null terminator (and excluding the null terminator itself).

```
#include <stdio.h>
#include <string.h>

#define SIZE20 20

int func()
{
    char s1[SIZE20] =  "abc";
    char s2[SIZE20] =  "abc";

    return memcmp(s1, s2, strlen(s1));
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `MEMCMP_STRINGS`
**Impact:** Medium

# Version History

**Introduced in R2017a**

## See Also

`Find defects (-checkers)` | `Memory comparison of padding data`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing byte reordering when transferring data

Different endianness of host and network

## Description

This defect occurs when you do not use a byte ordering function:

- Before sending data to a network socket.
- After receiving data from a network socket.

### Risk

Some system architectures implement little endian byte ordering (least significant byte first), and other systems implement big endian (most significant byte first). If the endianness of the sent data does not match the endianness of the receiving system, the value returned when reading the data is incorrect.

### Fix

After receiving data from a socket, use a byte ordering function such as `ntohl()`. Before sending data to a socket, use a byte ordering function such as `htonl()` .

## Examples

### Data Transferred Without Byte Reordering

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <byteswap.h>
#include <unistd.h>
#include <string.h>


unsigned int func(int sock, int server)
{
    unsigned int num;   /* assume int is 32-bits */
    if (server)
    {
        /* Server side */
        num = 0x17;
        /* Endianness of server host may not match endianness of network. */
        if (send(sock, (void *)&num, sizeof(num), 0) < (int)sizeof(num))
        {
            /* Handle error */
        }
        return 0;
    }
    else {
        /* Endianness of client host may not match endianness of network. */
        if (recv (sock, (void *)&num, sizeof(num), 0) < (int) sizeof(num))
        {
            /* Handle error */
        }

        /* Comparison may be inaccurate */
        if (num> 255)
        {
            return 255;
        }
        else
        {
            return num;
        }
    }
}
```

In this example, variable `num` is assigned hexadecimal value `0x17` and is sent over a network to the client from the server. If the server host is little endian and the network is big endian, `num` is transferred as `0x17000000`. The client then reads an incorrect value for `num` and compares it to a local numeric value.

**Correction — Use Byte Ordering Function**

Before sending `num` from the server host, use `htonl()` to convert from host to network byte ordering. Similarly, before reading `num` on the client host, use `ntohl()` to convert from network to host byte ordering.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <byteswap.h>
#include <unistd.h>
#include <string.h>

unsigned int func(int sock, int server)
{
    unsigned int num;   /* assume int is 32-bits */
    if (server)
    {
        /* Server side */
        num = 0x17;

        /* Convert to network byte order. */
        num = htonl(num);
        if (send(sock, (void *)&num, sizeof(num), 0) < (int)sizeof(num))
        {
            /* Handle error */
        }
        return 0;
    }
    else {
        if (recv (sock, (void *)&num, sizeof(num), 0) < (int) sizeof(num))
        {
            /* Handle error */
        }

        /* Convert to host byte order. */
        num = ntohl(num);
        if (num > 255)
        {
            return 255;
        }
        else
        {
            return num;
        }
    }
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `MISSING_BYTESWAP`
**Impact:** Medium

# Version History

**Introduced in R2017b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing null in string array

String does not terminate with null character

## Description

This defect occurs when a string does not have enough space to terminate with a null character `'\0'`.

This defect applies only for projects in C.

### Risk

A buffer overflow can occur if you copy a string to an array without assuming the implicit null terminator.

### Fix

If you initialize a character array with a literal, avoid specifying the array bounds.

```
char three[]  = "THREE";
```

The compiler automatically allocates space for a null terminator. In the preceding example, the compiler allocates sufficient space for five characters and a null terminator.

If the issue occurs after initialization, you might have to increase the size of the array by one to account for the null terminator.

In certain circumstances, you might want to initialize the character array with a sequence of characters instead of a string. In this situation, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Array size is too small

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE";
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

#### Correction — Increase Array Size

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

#### Correction — Change Initialization Method

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

## Result Information

**Group:** Programming
**Language:** C
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `MISSING_NULL_CHAR`
**Impact:** Low

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Misuse of a FILE object

Use of copy of FILE object

## Description

This defect occurs when:

- You dereference a pointer to a FILE object, including indirect dereference by using `memcmp()`.
- You modify an entire FILE object or one of its components through its pointer.
- You take the address of FILE object that was not returned from a call to an `fopen`-family function. No defect is raised if a macro defines the pointer as the address of a built-in FILE object, such as `#define ptr (&__stdout)`.

### Risk

In some implementations, the address of the pointer to a FILE object used to control a stream is significant. A pointer to a copy of a FILE object is interpreted differently than a pointer to the original object, and can potentially result in operations on the wrong stream. Therefore, the use of a copy of a FILE object can cause the software to stop responding, which an attacker might exploit in denial-of-service attacks.

### Fix

Do not make a copy of a FILE object. Do not use the address of a FILE object that was not returned from a successful call to an `fopen`-family function.

## Examples

### Copy of FILE Object Used in `fputs()`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

void fatal_error(void);

int func(void)
{
    /*'stdout' dereferenced and contents
        copied to 'my_stdout'. */
    FILE my_stdout = *stdout;

    /* Address of 'my_stdout' may not point to correct stream. */
    if (fputs("Hello, World!\n", &my_stdout) == EOF)
    {
        /* Handler error */
        fatal_error();
    }
    return 0;
}
```

In this example, FILE object `stdout` is dereferenced and its contents are copied to `my_stdout`. The contents of `stdout` might not be significant. `fputs()` is then called with the address of `my_stdout` as an argument. Because no call to `fopen()` or a similar function was made, the address of `my_stdout` might not point to the correct stream.

**Correction — Copy the FILE Object Pointer**

Declare `my_stdout` to point to the same address as `stdout` to ensure that you write to the correct stream when you call `fputs()`.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

void fatal_error(void);

int func(void)
{
    /* 'my_stdout' and 'stdout' point to the same object. */
    FILE *my_stdout = stdout;
    if (fputs("Hello, World!\n", my_stdout) == EOF)
    {
        /* Handler error */
        fatal_error();
    }
    return 0;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `FILE_OBJECT_MISUSE`
**Impact:** Low

# Version History

**Introduced in R2017b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Misuse of errno

errno incorrectly checked for error conditions

## Description

This defect occurs when you check errno for error conditions in situations where checking errno does not guarantee the absence of errors. In some cases, checking errno can lead to false positives.

For instance, you check errno following calls to the functions:

- fopen: If you follow the ISO Standard, the function might not set errno on errors.
- atof: If you follow the ISO Standard, the function does not set errno.
- signal: The errno value indicates an error only if the function returns the SIG_ERR error indicator.

### Risk

The ISO C Standard does not enforce that these functions set errno on errors. Whether the functions set errno or not is implementation-dependent.

To detect errors, if you check errno alone, the validity of this check also becomes implementation-dependent.

In some cases, the errno value indicates an error only if the function returns a specific error indicator. If you check errno before checking the function return value, you can see false positives.

### Fix

For information on how to detect errors, see the documentation for that specific function.

Typically, the functions return an out-of-band error indicator to indicate errors. For instance:

- fopen returns a null pointer if an error occurs.
- signal returns the SIG_ERR error indicator and sets errno to a positive value. Check errno only after you have checked the function return value.

## Examples

### Incorrectly Checking for errno After fopen Call

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    errno = 0;
    fileptr = fopen(temp_filename, "w+b");
    if (errno != 0) {
        if (fileptr != NULL) {
            (void)fclose(fileptr);
        }
        /* Handle error */
        fatal_error();
    }
    return fileptr;
}
```

In this example, errno is the first variable that is checked after a call to fopen. You might expect that fopen changes errno to a nonzero value if an error occurs. If you run this code with an implementation of fopen that does not set errno on errors, you might miss an error condition. In this situation, fopen can return a null pointer that escapes detection.

**Correction — Check Return Value of fopen After Call**

One possible correction is to only check the return value of fopen for a null pointer.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    fileptr = fopen(temp_filename, "w+b");
    if (fileptr == NULL) {
        fatal_error();
    }
    return fileptr;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** ERRNO_MISUSE
**Impact:** High

# Version History

**Introduced in R2017a**

## See Also

Find defects (-checkers) | Returned value of a sensitive function not checked | Errno not reset | Errno not checked | Unsafe conversion from string to numerical value

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Misuse of errno in a signal handler

You read `errno` after calling an `errno`-setting function in a signal handler

## Description

This defect occurs when you call one of these functions in a signal handler:

- `signal`: You call the `signal` function in a signal handler and then read the value of `errno`.

  For instance, the signal handler function `handler` calls `signal` and then calls `perror`, which reads `errno`.

  ```
  typedef void (*pfv)(int);

  void handler(int signum) {
    pfv old_handler = signal(signum, SIG_DFL);
    if (old_handler == SIG_ERR) {
      perror("SIGINT handler");
    }
  }
  ```
- `errno`-setting POSIX function: You call an `errno`-setting POSIX function in a signal handler but do not restore `errno` when returning from the signal handler.

  For instance, the signal handler function `handler` calls `waitpid`, which changes `errno`, but does not restore `errno` before returning.

  ```
  #include <stddef.h>
  #include <errno.h>
  #include <sys/wait.h>

  void handler(int signum) {
    int rc = waitpid(-1, NULL, WNOHANG);
    if (ECHILD != errno) {
    }
  }
  ```

### Risk

In each case that the checker flags, you risk relying on an indeterminate value of `errno`.

- `signal`: If the call to `signal` in a signal handler fails, the value of `errno` is indeterminate (see C11 Standard, Sec. 7.14.1.1). If you rely on a specific value of `errno`, you can see unexpected results.
- `errno`-setting POSIX function: An `errno`-setting function sets `errno` on failure. If you read `errno` after a signal handler is called and the signal handler itself calls an `errno`-setting function, you can see unexpected results.

### Fix

Avoid situations where you risk relying on an indeterminate value of `errno`.

- `signal`: After calling the `signal` function in a signal handler, do not read `errno` or use a function that reads `errno`.
- `errno`-setting POSIX function: Before calling an `errno`-setting function in a signal handler, save `errno` to a temporary variable. Restore `errno` from this variable before returning from the signal handler.

## Examples

### Reading errno After signal Call in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#define fatal_error() abort()

void handler(int signum) {
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        perror("SIGINT handler");
    }
}

int func(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
        fatal_error();
    }
    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
        fatal_error();
    }
    return 0;
}
```

In this example, the function `handler` is called to handle the `SIGINT` signal. In the body of `handler`, the `signal` function is called. Following this call, the value of `errno` is indeterminate. The checker raises a defect when the `perror` function is called because `perror` relies on the value of `errno`.

#### Correction — Avoid Reading errno After signal Call

One possible correction is to not read `errno` after calling the `signal` function in a signal handler. The corrected code here calls the `abort` function via the `fatal_error` macro instead of the `perror` function.

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#define fatal_error() abort()

void handler(int signum) {
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        fatal_error();
    }
}
```

```
int func(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
        fatal_error();
    }
    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
        fatal_error();
    }
    return 0;
}
```

### Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `SIG_HANDLER_ERRNO_MISUSE`
**Impact:** Medium

## Version History

**Introduced in R2018a**

### See Also

`Function called from signal handler not asynchronous-safe` | `Errno not checked` | `Errno not reset` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Misuse of narrow or wide character string

Narrow (wide) character string passed to wide (narrow) string function

## Description

This defect occurs when you pass a narrow character string to a wide string function, or a wide character string to a narrow string function.

**Misuse of narrow or wide character string** raises no defect on operating systems where narrow and wide character strings have the same size.

### Risk

Using a narrow character string with a wide string function, or vice versa, can result in unexpected or undefined behavior.

If you pass a wide character string to a narrow string function, you can encounter these issues:

- Data truncation. If the string contains null bytes, a copy operation using `strncpy()` can terminate early.
- Incorrect string length. `strlen()` returns the number of characters of a string up to the first null byte. A wide string can have additional characters after its first null byte.

If you pass a narrow character string to a wide string function, you can encounter this issue:

- Buffer overflow. In a copy operation using `wcsncpy()`, the destination string might have insufficient memory to store the result of the copy.

### Fix

Use the narrow string functions with narrow character strings. Use the wide string functions with wide character strings.

## Examples

### Passing Wide Character Strings to `strncpy()`

```
#include <string.h>
#include <wchar.h>

void func(void)
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    strncpy(wide_str2, wide_str1, 10);
}
```

In this example, `strncpy()` copies 10 wide characters from `wide_strt1` to `wide_str2`. If `wide_str1` contains null bytes, the copy operation can end prematurely and truncate the wide character string.

#### Correction — Use `wcsncpy()` to Copy Wide Character Strings

One possible correction is to use `wcsncpy()` to copy `wide_str1` to `wide_str2`.

```
#include <string.h>
#include <wchar.h>

void func(void)
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    wcsncpy(wide_str2, wide_str1, 10);
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `NARROW_WIDE_STR_MISUSE`
**Impact:** High

# Version History

**Introduced in R2018b**

## See Also

`Array access out of bounds` | `Destination buffer overflow in string manipulation` | `Invalid use of standard library routine` | `Invalid use of standard library string routine` | `Pointer access out of bounds` | `Unreliable cast of function pointer` | `Wrong allocated object size for cast` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Misuse of return value from non-reentrant standard function

Pointer to static buffer from previous call is used despite a subsequent call that modifies the buffer

## Description

This defect occurs when these events happen in this sequence:

1. You point to the buffer returned from a nonreentrant standard function such as `getenv` or `setlocale`.

   ```
   user = getenv("USER");
   ```
2. You call that nonreentrant standard function again.

   ```
   user2 = getenv("USER2");
   ```
3. You use or dereference the pointer from the first step expecting the buffer to remain unmodified since that step. In the meantime, the call in the second step has modified the buffer.

   For instance:

   ```
   var=*user;
   ```

In some cases, the defect might appear even if you do not call the `getenv` function a second time but simply return the pointer. For instance:

```
char* func() {
     user=getenv("USER");
     .
     .
     return user;
}
```

For information on which functions are covered by this defect, see documentation on nonreentrant standard functions.

### Risk

The C Standard allows nonreentrant functions such as `getenv` to return a pointer to a *static* buffer. Because the buffer is static, a second call to `getenv` modifies the buffer. If you continue to use the pointer returned from the first call past the second call, you can see unexpected results. The buffer that it points to no longer has values from the first call.

The defect appears even if you do not call `getenv` a second time but simply return the pointer. The reason is that someone calling your function might use the returned pointer *after* a second call to `getenv`. By returning the pointer from your call to `getenv`, you make your function unsafe to use.

The same rationale is true for other nonreentrant functions covered by this defect.

### Fix

After the first call to getenv, make a copy of the buffer that the returned pointer points to. After the second call to getenv, use this copy. Even if the second call modifies the buffer, your copy is untouched.

## Examples

### Return from getenv Used After Second Call to getenv

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");   /* First call */
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');

        if (user_name_from_home != NULL) {
            user = getenv("USER");   /* Second call */
            if ((user != NULL) &&
                (strcmp(user, user_name_from_home) == 0))
            {
                result = 1;
            }
        }
    }
    return result;
}
```

In this example, the pointer user_name_from_home is derived from the pointer home. home points to the buffer returned from the first call to getenv. Therefore, user_name_from_home points to a location in the same buffer.

After the second call to getenv, the buffer is modified. If you continue to use user_name_from_home, you can get unexpected results.

#### Correction — Make Copy of Buffer Before Second Call

If you want to access the buffer from the first call to getenv past the second call, make a copy of the buffer after the first call. One possible correction is to use the strdup function to make the copy.

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');
        if (user_name_from_home != NULL) {
            /* Make copy before second call */
            char *saved_user_name_from_home = strdup(user_name_from_home);
            if (saved_user_name_from_home != NULL) {
                user = getenv("USER");
                if ((user != NULL) &&
                    (strcmp(user, saved_user_name_from_home) == 0))
                {
                    result = 1;
                }
                free(saved_user_name_from_home);
            }
        }
    }
    return result;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** NON_REENTRANT_STD_RETURN
**Impact:** High

# Version History

**Introduced in R2017a**

## See Also

Find defects (-checkers) | Modification of internal buffer returned from nonreentrant standard function | Use of obsolete standard function

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Misuse of sign-extended character value

Data type conversion with sign extension causes unexpected behavior

## Description

This defect occurs when you convert a signed or plain `char` variable containing possible negative values to a wider integer data type (or perform an arithmetic operation that does the conversion) and then use the resulting value in one of these ways:

- For comparison with `EOF` (using == or `!=`)
- As array index
- As argument to a character-handling function in `ctype.h`, for instance, `isalpha()` or `isdigit()`

If you convert a signed `char` variable with a negative value to a wider type such as `int`, the sign bit is preserved (sign extension). This can lead to specific problems even in situations where you think you have accounted for the sign bit.

For instance, the signed `char` value of -1 can represent the character `EOF` (end-of-file), which is an invalid character. Suppose a `char` variable `var` acquires this value. If you treat `var` as a `char` variable, you might want to write special code to account for this invalid character value. However, if you perform an operation such as `var++` (involving integer promotion), it leads to the value 0, which represents a valid value `'\0'` by accident. You transitioned from an invalid to a valid value through the arithmetic operation.

Even for negative values other than -1, a conversion from signed `char` to signed `int` can lead to other issues. For instance, the signed `char` value -126 is equivalent to the `unsigned char` value 130 (corresponding to an extended character `'\202'`). If you convert the value from `char` to `int`, the sign bit is preserved. If you then cast the resulting value to `unsigned int`, you get an unexpectedly large value, 4294967170 (assuming 32-bit `int`). If your code expects the `unsigned char` value of 130 in the final `unsigned int` variable, you can see unexpected results.

The underlying cause of this issue is the sign extension during conversion to a wider type. Most architectures use two's complement representation for storing values. In this representation, the most significant bit indicates the sign of the value. When converted to a wider type, the conversion is done by copying this sign bit to all the leading bits of the wider type, so that the sign is preserved. For instance, the `char` value of -3 is represented as `11111101` (assuming 8-bit `char`). When converted to `int`, the representation is:

```
11111111 11111111 11111111  11111101
```

The value -3 is preserved in the wider type `int`. However, when converted to `unsigned int`, the value (4294967293) is no longer the same as the `unsigned char` equivalent of the original `char` value. If you are not aware of this issue, you can see unexpected results in your code.

### Risk

In the following cases, Bug Finder flags use of variables after a conversion from `char` to a wider data type or an arithmetic operation that implicitly converts the variable to a wider data type:

- *If you compare the variable value with EOF*:

  A char value of -1 can represent the invalid character EOF or the valid extended character value '\377' (corresponding to the unsigned char equivalent, 255). After a char variable is cast to a wider type such as int, because of sign extension, the char value -1, representing one of EOF or '\377' becomes the int value -1, representing only EOF. The unsigned char value 255 can no longer be recovered from the int variable. Bug Finder flags this situation so that you can cast the variable to unsigned char first (or avoid the char-to-int conversion or converting operation before comparison with EOF). Only then, a comparison with EOF is meaningful. See “Sign-Extended Character Value Compared with EOF” on page 11-125.
- *If you use the variable value as an array index*:

  After a char variable is cast to a wider type such as int, because of sign extension, all negative values retain their sign. If you use the negative values directly to access an array, you cause buffer overflow/underflow. Even when you account for the negative values, the way you account for them might result in incorrect elements being read from the array. See “Sign-Extended Character Value Used as Array Index” on page 11-126.
- *If you pass the variable value as argument to a character-handling function*:

  According to the C11 standard (Section 7.4), if you supply an integer argument that cannot be represented as unsigned char or EOF, the resulting behavior is undefined. Bug Finder flags this situation because negative char values after conversion can no longer be represented as unsigned char or EOF. For instance, the signed char value -126 is equivalent to the unsigned char value 130, but the signed int value -126 cannot be represented as unsigned char or EOF.

### Fix

Before conversion to a wider integer data type, cast the signed or plain char value explicitly to unsigned char.

If you use the char data type to not represent characters but simply as a smaller data type to save memory, your use of sign-extended char values might avoid the risks mentioned earlier. If so, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Examples

### Sign-Extended Character Value Compared with EOF

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
```

```
{
    int c = EOF;
    if (buf && *buf) {
        c = *buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

In this example, the function `parser` can traverse a string input `buf`. If a character in the string has the value -1, it can represent either `EOF` or the valid character value `'\377'` (corresponding to the `unsigned char` equivalent 255). When converted to the `int` variable `c`, its value becomes the integer value -1, which is always `EOF`. The later comparison with `EOF` will not detect if the value returned from `parser` is actually `EOF`.

#### Correction — Cast to `unsigned char` Before Conversion

One possible correction is to cast the plain `char` value to `unsigned char` before conversion to the wider `int` type. Only then can you test if the return value of `parser` is really `EOF`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = (unsigned char)*buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

### Sign-Extended Character Value Used as Array Index

```
#include <limits.h>
#include <stddef.h>
#include <stdio.h>

#define NUL '\0'
#define SOH 1    /* start of heading */
```

```
#define STX 2    /* start of text */
#define ETX 3    /* end of text */
#define EOT 4    /* end of transmission */
#define ENQ 5    /* enquiry */
#define ACK 6    /* acknowledge */

static const int ascii_table[UCHAR_MAX + 1] =
{
      [0]=NUL,[1]=SOH, [2]=STX, [3]=ETX, [4]=EOT, [5]=ENQ,[6]=ACK,
      /* ... */
      [126] = '~',
      /* ... */
      [130/*-126*/]='\202',
      /* ... */
      [255 /*-1*/]='\377'
};

int lookup_ascii_table(char c)
{
    int i;
    i = (c < 0 ? -c : c);
    return ascii_table[i];
}
```

In this example, the `char` variable `c` is converted to the `int` variable `i`. If `c` has negative values, they are converted to positive values before assignment to `i`. However, this conversion can lead to unexpected values when `i` is used as array index. For instance:

- If `c` has the value -1 representing the invalid character `EOF`, you want to probably treat this value separately. However, in this example, a value of `c` equal to -1 leads to a value of `i` equal to 1. The function `lookup_ascii_table` returns the value `ascii_table[1]` (or `SOH`) without the invalid character value `EOF` being accounted for.

  If you use the `char` data type to not represent characters but simply as a smaller data type to save memory, you need not worry about this issue.
- If `c` has a negative value, when assigned to `i`, its sign is reversed. However, if you access the elements of `ascii_table` through `i`, this sign reversal can result in unexpected values being read.

  For instance, if `c` has the value -126, `i` has the value 126. The function `lookup_ascii_table` returns the value `ascii_table[126]` (or `'~'`) but you probably expected the value `ascii_table[130]` (or `'\202'`).

#### Correction – Cast to `unsigned char`

To correct the issues, avoid the conversion from `char` to `int`. First, check `c` for the value `EOF`. Then, cast the value of the `char` variable `c` to `unsigned char` and use the result as array index.

```
#include <limits.h>
#include <stddef.h>
#include <stdio.h>

#define NUL '\0'
#define SOH 1    /* start of heading */
#define STX 2    /* start of text */
#define ETX 3    /* end of text */
```

```
#define EOT 4    /* end of transmission */
#define ENQ 5    /* enquiry */
#define ACK 6    /* acknowledge */

static const int ascii_table[UCHAR_MAX + 1] =
{
      [0]=NUL,[1]=SOH, [2]=STX, [3]=ETX, [4]=EOT, [5]=ENQ,[6]=ACK,
      /* ... */
      [126] = '~',
      /* ... */
      [130/*-126*/]='\202',
      /* ... */
      [255 /*-1*/]='\377'
};

int lookup_ascii_table(char c)
{
    int r = EOF;
    if (c != EOF) /* specific handling EOF, invalid character */
        r = ascii_table[(unsigned char)c]; /* cast to 'unsigned char' */
    return r;
}
```

### Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `CHARACTER_MISUSE`
**Impact:** Medium

## Version History

**Introduced in R2017a**

### See Also

`Find defects (-checkers)` | `Invalid use of standard library integer routine` | `Returned value of a sensitive function not checked` | `Errno not checked` | `Character value absorbed into EOF`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Misuse of structure with flexible array member

Memory allocation ignores flexible array member

## Description

This defect occurs when:

- You define an object with a flexible array member of unknown size at compilation time.
- You make an assignment between structures with a flexible array member without using `memcpy()` or a similar function.
- You use a structure with a flexible array member as an argument to a function and pass the argument by value.
- Your function returns a structure with a flexible array member.

A flexible array member has no array size specified and is the last element of a structure with at least two named members.

### Risk

If the size of the flexible array member is not defined, it is ignored when allocating memory for the containing structure. Accessing such a structure has undefined behavior.

### Fix

- Use `malloc()` or a similar function to allocate memory for a structure with a flexible array member.
- Use `memcpy()` or a similar function to copy a structure with a flexible array member.
- Pass a structure with a flexible array member as a function argument by pointer.

## Examples

### Structure Passed By Value to Function

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>


struct example_struct
{
    size_t num;
    int data[];
};

extern void arg_by_value(struct example_struct s);

void func(void)
```

```
{
    struct example_struct *flex_struct;
    size_t i;
    size_t array_size = 4;
    /* Dynamically allocate memory for the struct */
    flex_struct = (struct example_struct *)
        malloc(sizeof(struct example_struct) + sizeof(int) * array_size);
    if (flex_struct == NULL)
    {
        /* Handle error */
    }
    /* Initialize structure */
    flex_struct->num = array_size;
    for (i = 0; i < array_size; ++i)
    {
        flex_struct->data[i] = 0;
    }
    /* Handle structure */

    /* Argument passed by value. 'data' not
    copied to passed value. */
    arg_by_value(*flex_struct);

    /* Free dynamically allocated memory */
    free(flex_struct);
}
```

In this example, `flex_struct` is passed by value as an argument to `arg_by_value`. As a result, the flexible array member data is not copied to the passed argument.

**Correction — Pass Structure by Pointer to Function**

To ensure that all the members of the structure are copied to the passed argument, pass `flex_struct` to `arg_by_pointer` by pointer.

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>


struct example_struct
{
    size_t num;
    int data[];
};

extern void arg_by_pointer(struct example_struct *s);

void func(void)
{
    struct example_struct *flex_struct;
    size_t i;
    size_t array_size = 4;
    /* Dynamically allocate memory for the struct */
    flex_struct = (struct example_struct *)
        malloc(sizeof(struct example_struct) + sizeof(int) * array_size);
    if (flex_struct == NULL)
    {
        /* Handler error */
    }
    /* Initialize structure */
    flex_struct->num = array_size;
    for (i = 0; i < array_size; ++i)
    {
        flex_struct->data[i] = 0;
    }
    /* Handle structure */

    /* Structure passed by pointer */
    arg_by_pointer(flex_struct);

    /* Free dynamically allocated memory */
    free(flex_struct);
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `FLEXIBLE_ARRAY_MEMBER_STRUCT_MISUSE`
**Impact:** Low

# Version History

**Introduced in R2017b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Modification of internal buffer returned from non-reentrant standard function

Function attempts to modify internal buffer returned from a nonreentrant standard function

## Description

This defect occurs when the following happens:

- A nonreentrant standard function returns a pointer.
- You attempt to write to the memory location that the pointer points to.

Nonreentrant standard functions that return a non `const`-qualified pointer to an internal buffer include `getenv`, `getlogin`, `crypt`, `setlocale`, `localeconv`, `strerror` and others.

### Risk

Modifying the internal buffer that a nonreentrant standard function returns can cause the following issues:

- It is possible that the modification does not succeed or alters other internal data.

  For instance, `getenv` returns a pointer to an environment variable value. If you modify this value, you alter the environment of the process and corrupt other internal data.
- Even if the modification succeeds, it is possible that a subsequent call to the same standard function does not return your modified value.

  For instance, you modify the environment variable value that `getenv` returns. If another process, thread, or signal handler calls `setenv`, the modified value is overwritten. Therefore, a subsequent call to `getenv` does not return your modified value.

### Fix

Avoid modifying the internal buffer using the pointer returned from the function.

## Examples

### Modification of `getenv` Return Value

```
#include <stdlib.h>
#include <string.h>

void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        strncpy(env, "C", 1);
        printstr(env);
    }
}
```

In this example, the first argument of `strncpy` is the return value from a nonreentrant standard function `getenv`. The behavior can be undefined because `strncpy` modifies this argument.

#### Correction - Copy Return Value of getenv and Modify Copy

One possible solution is to copy the return value of `getenv` and pass the copy to the `strncpy` function.

```
#include <stdlib.h>
#include <string.h>
enum {
    SIZE20 = 20
};

void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        char env_cp[SIZE20];
        strncpy(env_cp, env, SIZE20);
        strncpy(env_cp, "C", 1);
        printstr(env_cp);
    }
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `WRITE_INTERNAL_BUFFER_RETURNED_FROM_STD_FUNC`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Overlapping assignment

Memory overlap between left and right sides of an assignment

## Description

This defect occurs when there is a memory overlap between the left and right sides of an assignment. For instance, a variable is assigned to itself or one member of a union is assigned to another.

### Risk

If the left and right sides of an assignment have memory overlap, the behavior is either redundant or undefined. For instance:

- Self-assignment such as `x=(int)(long)x;` is redundant unless `x` is `volatile`-qualified.
- Assignment of one union member to another causes undefined behavior.

  For instance, in the following code:

  - The result of the assignment `u1.a = u1.b` is undefined because `u1.b` is not initialized.
  - The result of the assignment `u2.b = u2.a` depends on the alignment and endianness of the implementation. It is not defined by C standards.

  ```
  union {
     char a;
     int b;
  }u1={'a'}, u2={'a'}; //'u1.a' and 'u2.a' are initialized

  u1.a = u1.b;
  u2.b = u2.a;
  ```

### Fix

Avoid assignment between two variables that have overlapping memory.

## Examples

### Assignment of Union Members

```
#include <string.h>

union Data {
    int i;
    float f;
};

int main( ) {
    union Data data;
    data.i = 0;
    data.f = data.i;
```

```
    return 0;
}
```

In this example, the variables `data.i` and `data.f` are part of the same `union` and are stored in the same location. Therefore, part of their memory storage overlaps.

### Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `OVERLAPPING_ASSIGN`
**Impact:** Low

## Version History

**Introduced in R2015b**

### See Also

`Find defects (-checkers)` | `Copy of overlapping memory`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Possible misuse of sizeof

Use of `sizeof` operator can cause unintended results

## Description

This defect occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

### Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

### Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

## Examples

### `sizeof` Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

#### Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `SIZEOF_MISUSE`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

#### External Websites

Linux man page for strncmp

# Possibly unintended evaluation of expression because of operator precedence rules

Operator precedence rules cause unexpected evaluation order in arithmetic expression

## Description

This defect occurs when an arithmetic expression result is possibly unintended because operator precedence rules dictate an evaluation order that you do not expect.

The defect highlights expressions of the form `x` *`op_1`* `y` *`op_2`* `z`. Here, *op_1* and *op_2* are operator combinations that commonly induce this error. For instance, `x == y | z`.

The checker does not flag all operator combinations. For instance, `x == y || z` is not flagged because you most likely intended to perform a logical OR between `x == y` and `z`. Specifically, the checker flags these combinations:

- && and `||`: For instance, `x || y && z` or `x && y || z`.
- Assignment and bitwise operations: For instance, `x = y | z`.
- Assignment and comparison operations: For instance, `x = y != z` or `x = y > z`.
- Comparison operations: For instance, `x > y > z` (except when one of the comparisons is an equality `x == y > z`).
- Shift and numerical operation: For instance, `x << y + 2`.
- Pointer dereference and arithmetic: For instance, `*p++`.

### Risk

The defect can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:
  - In the operation `*p++`, it is possible that you expect the dereferenced value to be incremented. However, the pointer `p` is incremented before the dereference.
  - In the operation `(x == y | z)`, it is possible that you expect `x` to be compared with `y | z`. However, the == operation happens before the `|` operation.

### Fix

See if the order of evaluation is what you intend. If not, apply parentheses to implement the evaluation order that you want.

For better readability of your code, it is good practice to apply parenthesis to implement an evaluation order even when operator precedence rules impose that order.

## Examples

### Expressions with Possibly Unintended Evaluation Order

```
int test(int a, int b, int c) {
    return(a & b == c);
}
```

In this example, the == operation happens first, followed by the & operation. If you intended the reverse order of operations, the result is not what you expect.

#### Correction — Parenthesis For Intended Order

One possible correction is to apply parenthesis to implement the intended evaluation order.

```
int test(int a, int b, int c) {
    return((a & b) == c);
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `OPERATOR_PRECEDENCE`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

### External Websites

C++ Operator Precedence

# Predefined macro used as an object

You use standard library macros such as `assert` and `errno` as objects

## Description

This defect occurs when you use certain identifiers in a way that requires an underlying object to be present. These identifiers are defined as macros. The C Standard does not allow you to redefine them as objects. You use the identifiers in such a way that macro expansion of the identifiers cannot occur.

For instance, you refer to an external variable `errno`:

```
extern int errno;
```

However, `errno` does not occur as a variable but a macro.

The defect applies to these macros: `assert`, `errno`, `math_errhandling`, `setjmp`, `va_arg`, `va_copy`, `va_end`, and `va_start`. The checker looks for the defect only in source files (not header files).

### Risk

The C11 Standard (Sec. 7.1.4) allows you to redefine most macros as objects. To access the object and not the macro in a source file, you do one of these:

- Redeclare the identifier as an external variable or function.
- For function-like macros, enclose the identifier name in parentheses.

If you try to use these strategies for macros that cannot be redefined as objects, an error occurs.

### Fix

Do not use the identifiers in such a way that a macro expansion is suppressed.

- Do not redeclare the identifiers as external variables or functions.
- For function-like macros, do not enclose the macro name in parentheses.

## Examples

### Use of `assert` as Function

```
#include<assert.h>
typedef void (*err_handler_func)(int);

extern void demo_handle_err(err_handler_func, int);

void func(int err_code) {
    extern void assert(int);
    demo_handle_err(&(assert), err_code);
}
```

In this example, the `assert` macro is redefined as an external function. When passed as an argument to `demo_handle_err`, the identifier `assert` is enclosed in parentheses, which suppresses use of the `assert` macro.

#### Correction — Use `assert` as Macro

One possible correction is to directly use the `assert` macro from `assert.h`. A different implementation of the function `demo_handle_err` directly uses the `assert` macro instead of taking the address of an `assert` function.

```
#include<assert.h>
void demo_handle_err(int err_code) {
    assert(err_code == 0);
}

void func(int err_code) {
    demo_handle_err(err_code);
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `MACRO_USED_AS_OBJECT`
**Impact:** Low

# Version History

**Introduced in R2018a**

## See Also

`MISRA C:2012 Rule 21.2` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Preprocessor directive in macro argument

You use a preprocessor directive in the argument to a function-like macro

## Description

This defect occurs when you use a preprocessor directive in the argument to a function-like macro or a function that might be implemented as a function-like macro.

For instance, a `#ifdef` statement occurs in the argument to a `memcpy` function. The `memcpy` function might be implemented as a macro.

```
memcpy(dest, src,
    #ifdef PLATFORM1
      12
    #else
      24
    #endif
  );
```

The checker flags similar usage in `printf` and `assert`, which can also be implemented as macros.

### Risk

During preprocessing, a function-like macro call is replaced by the macro body and the parameters are replaced by the arguments to the macro call (argument substitution). Suppose a macro `min()` is defined as follows.

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you call `min(1,2)`, it is replaced by the body `((X) < (Y) ? (X) : (Y))`. `X` and `Y` are replaced by 1 and 2.

According to the C11 Standard (Sec. 6.10.3), if the list of arguments to a function-like macro itself has preprocessing directives, the argument substitution during preprocessing is undefined.

### Fix

To ensure that the argument substitution happens in an unambiguous manner, use the preprocessor directives outside the function-like macro.

For instance, to execute `memcpy` with different arguments based on a `#ifdef` directive, call `memcpy` multiple times within the `#ifdef` directive branches.

```
#ifdef PLATFORM1
    memcpy(dest, src, 12);
#else
    memcpy(dest, src, 24);
#endif
```

## Examples

### Directives in Function-Like Macros

```
#include <stdio.h>

#define print(A) printf(#A)

void func(void) {
    print(
#ifdef SW
          "Message 1"
#else
          "Message 2"
#endif
         );
}
```

In this example, the preprocessor directives `#ifdef` and `#endif` occur in the argument to the function-like macro `print()`.

#### Correction — Use Directives Outside Macro

One possible correction is to use the function-like macro multiple times in the branches of the `#ifdef` directive.

```
#include <stdio.h>

#define print(A) printf(#A)

void func(void) {
#ifdef SW
        print("Message 1");
#else
        print("Message 2");
#endif
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `PRE_DIRECTIVE_MACRO_ARG`
**Impact:** Low

# Version History

**Introduced in R2018a**

## See Also

`MISRA C:2012 Rule 20.6` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"

“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Qualifier removed in conversion

Variable qualifier is lost during conversion

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This defect occurs during a pointer conversion when one pointer has a qualifier and the other does not. For example, when converting from a `const int*` to an `int*`, the conversion removes the `const` qualifier.

This defect applies only for projects in C.

### Risk

Qualifiers such as `const` or `volatile` in a pointer declaration:

```
const int* ptr;
```

imply that the underlying object is `const` or `volatile`. These qualifiers act as instructions to the compiler. For instance, a `const` object is not supposed to be modified in the code and a `volatile` object is not supposed to be optimized away by the compiler.

If a second pointer points to the same object but does not use the same qualifier, the qualifier on the first pointer is no longer valid. For instance, if a `const int*` pointer and an `int*` pointer point to the same object, you can modify the object through the second pointer and violate the contract implied by the `const` qualifier in the first pointer.

### Fix

If you intend to convert from one pointer to another, declare both pointers with the same qualifiers.

## Examples

### Cast of Character Pointers

```
void read(char *);

void implicit_cast(void) {
    const char  cc, *pcc = &cc;
    char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

During the assignment to the character `q`, the variables, `cc` and `pcc`, are converted from `const char` to `char`. The `const` qualifier is removed during the conversion causing a defect.

#### Correction — Add Qualifiers

One possible correction is to add the same qualifiers to the new variables. In this example, changing `q` to a `const char` fixes the defect.

```
void read(char *);

void implicit_cast(void) {
    const char  cc, *pcc = &cc;
    const char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

#### Correction — Remove Qualifiers

One possible correction is to remove the qualifiers in the converted variable. In this example, removing the `const` qualifier from the `cc` and `pcc` initialization fixes the defect.

```
void read(char *);

void implicit_basic_cast(void) {
    char  cc, *pcc = &cc;
    char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

## Result Information

**Group:** Programming
**Language:** C
**Default:** Off
**Command-Line Syntax:** `QUALIFIER_MISMATCH`
**Impact:** Low

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Return from computational exception signal handler

Undefined behavior when signal handler returns normally from program error

## Description

This defect occurs when a signal handler returns after catching a computational exception signal SIGFPE, SIGILL, or SIGSEGV.

### Risk

A signal handler that returns normally from a computational exception is undefined behavior. Even if the handler attempts to fix the error that triggered the signal, the program can behave unexpectedly.

### Fix

Check the validity of the values of your variables before the computation to avoid using a signal handler to catch exceptions. If you cannot avoid a handler to catch computation exception signals, call abort(), quick_exit(), or _Exit() in the handler to stop the program.

## Examples

### Signal Handler Return from Division by Zero

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

static volatile sig_atomic_t denom;
/* Declare signal handler to catch division by zero
computation error. */
void sig_handler(int s)
{
    int s0 = s;
    if (denom == 0)
    {
        denom = 1;
    }
    /* Normal return from computation exception
    signal */
    return;
}


long func(int v)
{
    denom = (sig_atomic_t)v;

        if (signal(SIGFPE, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }

    long result = 100 / (long)denom;
    return result;
}
```

In this example, `sig_handler` is declared to handle a division by zero computation error. The handler changes the value of `denom` if it is zero and returns, which is undefined behavior.

**Correction — Call `abort()` to Terminate Program**

After catching a computational exception, call `abort()` from `sig_handler` to exit the program without further error.

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

static volatile sig_atomic_t denom;
/* Declare signal handler to catch division by zero
computation error. */

void sig_handler(int s)
{
    int s0 = s;
    /* call to abort() to exit the program */
    abort();
}

long func(int v)
{
    denom = (sig_atomic_t)v;

        if (signal(SIGFPE, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }

    long result = 100 / (long)denom;
    return result;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `SIG_HANDLER_COMP_EXCP_RETURN`
**Impact:** Low

## Version History

**Introduced in R2017b**

## See Also

`Function called from signal handler not asynchronous-safe` | `Function called from signal handler not asynchronous-safe (strict)` | `Signal call from within signal handler` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Shared data access within signal handler

Access or modification of shared data causes inconsistent state

## Description

This defect occurs when you access or modify a shared object inside a signal handler.

### Risk

When you define a signal handler function to access or modify a shared object, the handler accesses or modifies the shared object when it receives a signal. If another function is already accessing the shared object, that function causes a race condition and can leave the data in an inconsistent state.

### Fix

To access or modify shared objects inside a signal handler, check that the objects are lock-free atomic, or, if they are integers, declare them as `volatile sig_atomic_t`.

## Examples

### `int` Variable Access in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* declare global variable. */
int e_flag;

void sig_handler(int signum)
{
    /* Signal handler accesses variable that is not
    of type volatile sig_atomic_t. */
    e_flag = signum;
}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

In this example, sig_handler accesses e_flag, a variable of type int. A concurrent access by another function can leave e_flag in an inconsistent state.

#### Correction — Declare Variable of Type volatile sig_atomic_t

Before you access a shared variable from a signal handler, declare the variable with type volatile sig_atomic_t instead of int. You can safely access variables of this type asynchronously.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* Declare variable of type volatile sig_atomic_t. */
volatile sig_atomic_t e_flag;
void sig_handler(int signum)
{
    /* Use variable of proper type inside signal handler. */
    e_flag = signum;

}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** SIG_HANDLER_SHARED_OBJECT
**Impact:** Medium

# Version History

**Introduced in R2017b**

## See Also

Function called from signal handler not asynchronous-safe | Signal call from within signal handler | Find defects (-checkers)

**Topics**

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Side effect in arguments to unsafe macro

Macro contains arguments that can be evaluated multiple times or not evaluated

## Description

This defect occurs when you call an unsafe macro with an expression that has a side effect.

- *Unsafe macro*: When expanded, an unsafe macro evaluates its arguments multiple times or does not evaluate its argument at all.

  For instance, the `ABS` macro evaluates its argument `x` twice.

  ```
  #define ABS(x) (((x) < 0) ? -(x) : (x))
  ```
- *Side effect*: When evaluated, an expression with a side effect modifies at least one of the variables in the expression.

  For instance, `++n` modifies `n`, but `n+1` does not modify `n`.

  The checker does not consider side effects in nested macros. The checker also does not consider function calls or volatile variable access as side effects.

### Risk

If you call an unsafe macro with an expression that has a side effect, the expression is evaluated multiple times or not evaluated at all. The side effect can occur multiple times or not occur at all, causing unexpected behavior.

For instance, in the call `MACRO(++n)`, you expect only one increment of the variable `n`. If `MACRO` is an unsafe macro, the increment happens more than once or does not happen at all.

The checker flags expressions with side effects in the `assert` macro because the `assert` macro is disabled in non-debug mode. To compile in non-debug mode, you define the `NDEBUG` macro during compilation. For instance, in GCC, you use the flag `-DNDEBUG`.

### Fix

Evaluate the expression with a side effect in a separate statement, and then use the result as a macro argument.

For instance, instead of:

```
MACRO(++n);
```

perform the operation in two steps:

```
++n;
MACRO(n);
```

Alternatively, use an inline function instead of a macro. Pass the expression with side effect as argument to the inline function.

The checker considers modifications of a local variable defined only in the block scope of a macro body as a side effect. This defect cannot happen since the variable is visible only in the macro body. If you see a defect of this kind, ignore the defect.

## Examples

### Macro Argument with Side Effects

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  int m = ABS(++n);

  /* ... */
}
```

In this example, the ABS macro evaluates its argument twice. The second evaluation can result in an unintended increment.

#### Correction — Separate Evaluation of Expression from Macro Usage

One possible correction is to first perform the increment, and then pass the result to the macro.

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  ++n;
  int m = ABS(n);

  /* ... */
}
```

#### Correction — Evaluate Expression in Inline Function

Another possible correction is to evaluate the expression in an inline function.

```
static inline int iabs(int x) {
  return (((x) < 0) ? -(x) : (x));
}

void func(int n) {
  /* Validate that n is within the desired range */

int m = iabs(++n);

  /* ... */
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** SIDE_EFFECT_IN_UNSAFE_MACRO_ARG
**Impact:** Medium

## Version History

**Introduced in R2018b**

## See Also

`Side effect of expression ignored` | `Stream argument with possibly unintended side effects` | `MISRA C:2012 Rule 13.2` | `MISRA C:2012 Rule 13.3` | `MISRA C:2012 Rule 13.4` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Side effect of expression ignored

`sizeof`, `_Alignof`, or `_Generic` operates on expression with side effect

## Description

This defect occurs when the `sizeof`, `_Alignof`, or `_Generic` operator operates on an expression with a side effect. When evaluated, an expression with side effect modifies at least one of the variables in the expression.

For instance, the defect checker does not flag `sizeof(n+1)` because `n+1` does not modify `n`. The checker flags `sizeof(n++)` because `n++` is intended to modify `n`.

The check also applies to the C++ operator `alignof` and its C extensions, `__alignof__` and `__typeof__`.

### Risk

The expression in a `_Alignof` or `_Generic` operator is not evaluated. The expression in a `sizeof` operator is evaluated only if it is required for calculating the size of a variable-length array, for instance, `sizeof(a[n++])`.

When an expression with a side effect is not evaluated, the variable modification from the side effect does not happen. If you rely on the modification, you can see unexpected results.

### Fix

Evaluate the expression with a side effect in a separate statement, and then use the result in a `sizeof`, `_Alignof`, or `_Generic` operator.

For instance, instead of:

```
a = sizeof(n++);
```

perform the operation in two steps:

```
n++;
a = sizeof(n);
```

The checker considers a function call as an expression with a side effect. Even if the function does not have side effects now, it might have side effects on later additions. The code is more maintainable if you call the function outside the `sizeof` operator.

## Examples

### Increment Operator in `sizeof`

```
#include <stdio.h>

void func(void) {
    unsigned int a = 1U;
    unsigned int b = (unsigned int)sizeof(++a);
    printf ("%u, %u\n", a, b);
}
```

In this example, `sizeof` operates on `++a`, which is intended to modify `a`. Because the expression is not evaluated, the modification does not happen. The `printf` statement shows that `a` still has the value `1`.

#### Correction — Perform Increment Outside `sizeof`

One possible correction is to perform the increment first, and then provide the result to the `sizeof` operator.

```
#include <stdio.h>

void func(void) {
    unsigned int a = 1U;
    ++a;
    unsigned int b = (unsigned int)sizeof (a);
    printf ("%u, %u\n", a, b);
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `SIDE_EFFECT_IGNORED`
**Impact:** Low

# Version History

**Introduced in R2018a**

## See Also

`MISRA C:2012 Rule 13.6` | `Redundant expression in sizeof operand` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Signal call from within signal handler

Nonpersistent signal handler calling `signal()` in Windows system causes race condition

## Description

This defect occurs when you call the function `signal()` from a signal handler on Windows platforms.

The defect is detected only if you specify a Visual Studio compiler. See `Compiler (-compiler)`.

### Risk

The function `signal()` associates a signal with a signal handler function. On platforms such as Windows, which removes this association after receiving the signal, you might call the function `signal()` again *within the signal handler* to re-establish the association.

However, this attempt to make a signal handler persistent is prone to race conditions. On Windows platforms, from the time the signal handler begins execution to when the `signal` function is called again, it is the default signal handling, `SIG_DFL`, that is active. If a second signal is received within this time window, you see the default signal handling and not the custom signal handler, but you might expect otherwise.

### Fix

Do not call `signal()` from a signal handler on Windows platforms.

## Examples

### `signal()` Called from Signal Handler

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;

void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;

    /* Call signal() to reestablish sig_handler
    upon receiving SIG_ERR. */

    if (signal(s0, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
}
```

```
void func(void)
{
        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
  /* more code */
}
```

In this example, the definition of `sig_handler()` includes a call to `signal()` when the handler catches `SIG_ERR`. On Windows platforms, signal handlers are nonpersistent. This code can result in a race condition.

The issue is detected only if you specify a compiler such as `visual15.x` for the analysis.

**Correction — Do Not Call `signal()` from Signal Handler**

Avoid attempting to make a signal handler persistent on Windows. If your code requires the use of a persistent signal handler on a Windows platform, use a persistent signal handler after performing a thorough risk analysis.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;


void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;
    /* No call to signal() */
}

int main(void)
{

        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code

**Command-Line Syntax:** `SIG_HANDLER_CALLING_SIGNAL`
**Impact:** Medium

## Version History

**Introduced in R2017b**

## See Also

`Find defects (-checkers)` | `Function called from signal handler not asynchronous-safe` | `Return from computational exception signal handler` | `Shared data access within signal handler`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Standard function call with incorrect arguments

Argument to a standard function does not meet requirements for use in the function

## Description

This defect occurs when the arguments to certain standard functions do not meet the requirements for their use in the functions.

For instance, the arguments to these functions can be invalid in the following ways.

| Function Type | Situation | Risk | Fix |
|---|---|---|---|
| String manipulation functions such as `strlen` and `strcpy` | The pointer arguments do not point to a `NULL`-terminated string. | The behavior of the function is undefined. | Pass a `NULL`-terminated string to string manipulation functions. |
| File handling functions in `stdio.h` such as `fputc` and `fread` | The `FILE*` pointer argument can have the value `NULL`. | The behavior of the function is undefined. | Test the `FILE*` pointer for `NULL` before using it as function argument. |
| File handling functions in `unistd.h` such as `lseek` and `read` | The file descriptor argument can be -1. | The behavior of the function is undefined.<br>Most implementations of the `open` function return a file descriptor value of -1. In addition, they set `errno` to indicate that an error has occurred when opening a file. | Test the return value of the `open` function for -1 before using it as argument for `read` or `lseek`.<br>If the return value is -1, check the value of `errno` to see which error has occurred. |
| | The file descriptor argument represents a closed file descriptor. | The behavior of the function is undefined. | Close the file descriptor only after you have completely finished using it. Alternatively, reopen the file descriptor before using it as function argument. |
| Directory name generation functions such as `mkdtemp` and `mkstemps` | The last six characters of the string template are not `XXXXXX`. | The function replaces the last six characters with a string that makes the file name unique. If the last six characters are not `XXXXXX`, the function cannot generate a unique enough directory name. | Test if the last six characters of a string are `XXXXXX` before using the string as function argument. |

| Function Type | Situation | Risk | Fix |
|---|---|---|---|
| Functions related to environment variables such as `getenv` and `setenv` | The string argument is `""`. | The behavior is implementation-defined. | Test the string argument for `""` before using it as `getenv` or `setenv` argument. |
| | The string argument terminates with an equal sign, =. For instance, `"C="` instead of `"C"`. | The behavior is implementation-defined. | Do not terminate the string argument with =. |
| String handling functions such as `strtok` and `strstr` | • `strtok`: The delimiter argument is `""`.<br>• `strstr`: The search string argument is `""`. | Some implementations do not handle these edge cases. | Test the string for `""` before using it as function argument. |

### Fix

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### `NULL` Pointer Passed as `strnlen` Argument

```
#include <string.h>
#include <stdlib.h>

enum {
    SIZE10 = 10,
    SIZE20 = 20
};

int func() {
    char* s = NULL;
    return strnlen(s, SIZE20);
}
```

In this example, a `NULL` pointer is passed as `strnlen` argument instead of a `NULL`-terminated string.

Before running analysis on the code, specify a GNU compiler. See `Compiler (-compiler)`.

#### Correction — Pass NULL-terminated String

Pass a `NULL`-terminated string as the first argument of `strnlen`.

```
#include <string.h>
#include <stdlib.h>

enum {
    SIZE10 = 10,
    SIZE20 = 20
};

int func() {
    char* s = "";
    return strnlen(s, SIZE20);
}
```

### Result Information

**Group:** Programming<br>
**Language:** C | C++<br>
**Default:** On for handwritten code, off for generated code<br>
**Command-Line Syntax:** `STD_FUNC_ARG_MISMATCH`<br>
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Find defects (-checkers)`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Stream argument with possibly unintended side effects

Stream argument side effects occur more than once

## Description

This defect occurs when you call `getc()`, `putc()`, `getwc()`, or `putwc()` with a stream argument that has side effects.

**Stream argument with possibly unintended side effects** considers the following as stream side effects:

- Any assignment of a variable of a stream, such as `FILE *`, or any assignment of a variable of a deeper stream type, such as an array of `FILE *`.
- Any call to a function that manipulates a stream or a deeper stream type.

The number of defects raised corresponds to the number of side effects detected. When a stream argument is evaluated multiple times in a function implemented as a macro, a defect is raised for each evaluation that has a side effect.

A defect is also raised on functions that are not implemented as macros but that can be implemented as macros on another operating system.

### Risk

If the function is implemented as an unsafe macro, the stream argument can be evaluated more than once, and the stream side effect happens multiple times. For instance, a stream argument calling `fopen()` might open the same file multiple times, which is unspecified behavior.

### Fix

To ensure that the side effect of a stream happens only once, use a separate statement for the stream argument.

## Examples

### Stream Argument of `getc()` Has Side Effect `fopen()`

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

#define fatal_error() abort()

const char* myfile = "my_file.log";

void func(void)
{
    int c;
    FILE* fptr;
    /* getc() has stream argument fptr with
    * 2 side effects: call to fopen(), and assignment
    * of fptr
    */
    c = getc(fptr = fopen(myfile, "r"));
    if (c == EOF) {
        /* Handle error */
        (void)fclose(fptr);
        fatal_error();
    }
    if (fclose(fptr) == EOF) {
        /* Handle error */
        fatal_error();
    }
}

void main(void)
{
    func();

}
```

In this example, `getc()` is called with stream argument `fptr`. The stream argument has two side effects: the call to `fopen()` and the assignment of `fptr`. If `getc()` is implemented as an unsafe macro, the side effects happen multiple times.

**Correction — Use Separate Statement for `fopen()`**

One possible correction is to use a separate statement for `fopen()`. The call to `fopen()` and the assignment of `fptr` happen in this statement so there are no side effects when you pass `fptr` to `getc()`.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

#define fatal_error() abort()


const char* myfile = "my_file.log";

void func(void)
{
    int c;
    FILE* fptr;

    /* Separate statement for fopen()
    * before call to getc()
    */
    fptr = fopen(myfile, "r");
    if (fptr == NULL) {
        /* Handle error */
        fatal_error();
    }
    c = getc(fptr);
    if (c == EOF) {
        /* Handle error */
        (void)fclose(fptr);
        fatal_error();
    }
    if (fclose(fptr) == EOF) {
        /* Handle error */
        fatal_error();
    }
}

void main(void)
{
    func();

}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `STREAM_WITH_SIDE_EFFECT`
**Impact:** Low

# Version History

**Introduced in R2018a**

## See Also

`Find defects (-checkers)` | `Returned value of a sensitive function not checked` | `Opening previously opened resource` | `Standard function call with incorrect arguments`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Too many va_arg calls for current argument list

Number of calls to va_arg exceeds number of arguments passed to variadic function

## Description

This defect occurs when the number of calls to va_arg exceeds the number of arguments passed to the corresponding variadic function. The analysis raises a defect only when the variadic function is called.

**Too many va_arg calls for current argument list** does not raise a defect when:

- The number of calls to va_arg inside the variadic function is indeterminate. For example, if the calls are from an external source.
- The va_list used in va_arg is invalid.

### Risk

When you call va_arg and there is no next argument available in va_list, the behavior is undefined. The call to va_arg might corrupt data or return an unexpected result.

### Fix

Ensure that you pass the correct number of arguments to the variadic function.

## Examples

### No Argument Available When Calling va_arg

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/
int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {
/* No further argument available
* in va_list when calling va_arg
*/

            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100);

}
```

In this example, the named argument and only one variadic argument are passed to `variadic_func()` when it is called inside `func()`. On the second call to `va_arg`, no further variadic argument is available in `ap` and the behavior is undefined.

#### Correction — Pass Correct Number of Arguments to Variadic Function

One possible correction is to ensure that you pass the correct number of arguments to the variadic function.

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/

int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {

/* The correct number of arguments is
* passed to va_list when variadic_func()
* is called inside func()
*/
            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100, 200);

}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code

**Command-Line Syntax:** `TOO_MANY_VA_ARG_CALLS`<br>
**Impact:** Medium

## Version History

**Introduced in R2018a**

## See Also

`Find defects (-checkers)` | `Invalid va_list argument` | `Incorrect data type passed to va_arg`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Typedef mismatch

Mismatch between typedef statements

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

This defect occurs when typedef statements lead to conflicting underlying types for one of these data types:

- size_t
- ssize_t
- wchar_t
- ptrdiff_t

### Risk

If you change the underlying type of size_t, ssize_t, wchar_t, or ptrdiff_t, you have inconsistent definitions of the same type. Compilation units with different include paths can potentially use different-sized types causing conflicts in your program.

For example, say that you define a function in one compilation unit that redefines size_t as unsigned long. But in another compilation unit that uses the size_t definition from <stddef.h>, you use the same function as an extern declaration. Your program will encounter a mismatch between the function declaration and function definition.

### Fix

Use consistent type definitions. For example:

- Remove custom type definitions for these fundamental types. Only use system definitions.
- Use the same size for all compilation units. Move your typedef to a shared header file.

## Examples

### Two Definitions of size_t

file1.c:

```
#include <stddef.h>

void func1()
{
    size_t var = 0;
    /*... more code ... */
}
```

file2.c:

```
typedef unsigned char size_t;

void func2()
{
    size_t var = 0;
    /*... more code ... */
}
```

In this example, Polyspace flags the definition of `size_t` in `file2.c` as a defect. This definition is a `typedef` mismatch because another file in your project, `file1.c`, includes `stddef.h`, which defines `size_t` as unsigned long.

### Correction — Use System Definition

One possible correction is to use the system definition of `size_t` in `stddef.h` to avoid conflicting type definitions.

`file1.c`:

```
#include <stddef.h>

void func1()
{
    size_t var = 0;
    /*... more code ... */
}
```

`file2.c`:

```
#include <stddef.h>

void func2()
{
    size_t var = 0;
    /*... more code ... */
}
```

### Correction — Use Shared Header File

One possible correction is to use a shared header file to store your type definition that gets included in both files.

`types.h`:

```
typedef unsigned char size_t;
```

`file1.c`:

```
#include "types.h"

void func1()
{
    size_t var = 0;
    /*... more code ... */
}
```

`file2.c`:

```
#include "types.h"
```

```
void func2()
{
    size_t var = 0;
    /*... more code ... */
}
```

### Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `TYPEDEF_MISMATCH`
**Impact:** High

## Version History

**Introduced in R2016b**

### See Also

`Declaration mismatch` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Universal character name from token concatenation

You create a universal character name by joining tokens with ## operator

## Description

This defect occurs when two preprocessing tokens joined with a ## operator create a universal character name. A universal character name begins with `\u` or `\U` followed by hexadecimal digits. It represents a character not found in the basic character set.

For instance, you form the character `\u0401` by joining two tokens:

```
#define assign(uc1, uc2, val) uc1##uc2 = val
...
assign(\u04, 01, 4);
```

### Risk

The C11 Standard (Sec. 5.1.1.2) states that if a universal character name is formed by token concatenation, the behavior is undefined.

### Fix

Use the universal character name directly instead of producing it through token concatenation.

## Examples

### Universal Character Name from Token Concatenation

```
#define assign(uc1, uc2, val) uc1##uc2 = val

int func(void) {
    int \u0401 = 0;
    assign(\u04, 01, 4);
    return \u0401;
}
```

In this example, the `assign` macro, when expanded, joins the two tokens `\u04` and `01` to form the universal character name `\u0401`.

#### Correction — Use Universal Character Name Directly

One possible correction is to use the universal character name `\u0401` directly. The correction redefines the `assign` macro so that it does not join tokens.

```
#define assign(ucn, val) ucn = val

int func(void) {
    int \u0401 = 0;
    assign(\u0401, 4);
    return \u0401;
}
```

### Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `PRE_UCNAME_JOIN_TOKENS`
**Impact:** Low

## Version History

**Introduced in R2018a**

### See Also

`MISRA C:2012 Rule 20.10` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unnamed namespace in header file

Header file contains unnamed namespace leading to multiple definitions

## Description

This defect occurs when an unnamed namespace is used in a header file, which can lead to multiple definitions of objects in the namespace.

### Risk

According to the C++ standard, names in an unnamed namespace, for instance, `aVar`:

```
namespace {
   int aVar;
}
```

have internal linkage by default. If a header file contains an unnamed namespace, each translation unit with a source file that `#include`-s the header file defines its own instance of objects in the namespace. The multiple definitions are probably not what you intended and can lead to unexpected results, unwanted excess memory usage, or inadvertently violating the one-definition rule.

### Fix

Specify names for namespaces in header files or avoid using namespaces in header files.

## Examples

### Unexpected Results from Unnamed Namespaces in Header Files

Header File: `aHeader.h`

```
namespace {
   int aVar;
}
```

First source file: `aSource.cpp`

```
#include "aHeader.h"
#include <iostream>

void setVar(int arg) {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = arg;
    std::cout << "Value set at: " << aVar << std::endl;
}
```

Second source file: `anotherSource.cpp`

```
#include "aHeader.h"
#include <iostream>

extern void setVar(int);
```

```
void resetVar() {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = 0;
    std::cout << "Value set at: 0" << std::endl;
}

void main() {
    setVar(1);
    resetVar();
}
```

In this example, the unnamed namespace leads to two definitions of `aVar` in the translation unit from `aSource.cpp` and the translation unit from `anotherSource.cpp`. The two definitions lead to possible unexpected output:

```
Current value: 0
Value set at: 1
Current value: 0
Value set at: 0
```

**Correction – Avoid the Unnamed Namespace**

One possible correction is to avoid a namespace in the header file.

Header File: `aHeader.h`

```
extern int aVar;
```

First source file: `aSource.cpp`

```
#include "aHeader.h"
#include <iostream>

void setVar(int arg) {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = arg;
    std::cout << "Value set at: " << aVar << std::endl;
}
```

Second source file: `anotherSource.cpp`

```
#include "aHeader.h"
#include <iostream>

extern void setVar(int);
int aVar;

void resetVar() {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = 0;
    std::cout << "Value set at: 0" << std::endl;
}

void main() {
    setVar(1);
    resetVar();
}
```

You now see the expected sequence in the output:

```
Current value: 0
Value set at: 1
Current value: 1
Value set at: 0
```

## Result Information

**Group:** Programming
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `UNNAMED_NAMESPACE_IN_HEADER`
**Impact:** Medium

# Version History

**Introduced in R2019b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unsafe conversion between pointer and integer

Misaligned or invalid results from conversions between pointer and integer types

## Description

This defect occurs when you convert between a pointer type, such as `intptr_t`, or `uintprt_t`, and an integer type, such as `enum`, `ptrdiff_t`, or `pid_t`, or vice versa.

### Risk

The mapping between pointers and integers is not always consistent with the addressing structure of the environment.

Converting from pointers to integers can create:

- Truncated or out of range integer values.
- Invalid integer types.

Converting from integers to pointers can create:

- Misaligned pointers or misaligned objects.
- Invalid pointer addresses.

### Fix

Where possible, avoid pointer-to-integer or integer-to-pointer conversions. If you want to convert a `void` pointer to an integer, so that you do not change the value, use types:

- C99 — `intptr_t` or `uintptr_t`
- C90 — `size_t` or `ssize_t`

## Examples

### Integer to Pointer Conversions

```
 unsigned int *badintptrcast(void)
{
    unsigned int *ptr0 = (unsigned int *)0xdeadbeef;
    char *ptr1 = (char *)0xdeadbeef;
    unsigned int *explicit_ptr = reinterpret_cast<unsigned int*>(0xdeadbeef);
    return (unsigned int *)(ptr0 - (unsigned int *)ptr1);
}
```

In this example, there are four conversions, three unsafe conversions and one safe conversion.

- The conversion of `0xdeadbeef` to an `unsigned int*` causes alignment issues for the pointer. Polyspace flags this conversion.
- The conversion of `0xdeadbeef` to a `char *` is not flagged because there are no alignment issues for `char`.

- The explicit reinterpret_cast of 0xdeadbeef to an unsigned int* causes alignment issues for the pointer. Polyspace flags this conversion.
- The conversion in the return casts ptrdiff_t to a pointer. This pointer might point to an invalid address. Polyspace flags this conversion.

#### Correction — Use intptr_t

One possible correction is to use intptr_t types to store the pointer address 0xdeadbeef. Also, you can change the second pointer to an integer offset so that there is no longer a conversion from ptrdiff_t to a pointer.

```
#include <stdint.h>

unsigned int *badintptrcast(void)
{
    intptr_t iptr0 = (intptr_t)0xdeadbeef;
    int offset = 0;
    return (unsigned int *)(iptr0 - offset);
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** BAD_INT_PTR_CAST
**Impact:** Medium

# Version History

**Introduced in R2016b**

## See Also

Find defects (-checkers)

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Unsafe conversion from string to numerical value

String to number conversion without validation checks

## Description

This defect occurs when you perform conversions from strings to integer or floating-point values and your conversion method does not include robust error handling.

### Risk

Converting a string to numerical value can cause data loss or misinterpretation. Without validation of the conversion or error handling, your program continues with invalid values.

### Fix

- Add additional checks to validate the numerical value.
- Use a more robust string-to-numeric conversion function such as `strtol`, `strtoll`, `strtoul`, or `strtoull`.

## Examples

### Conversion With `atoi`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char* argv1)
{
    int s = 0;
    if (demo_check_string_not_empty(argv1))
    {
        s = atoi(argv1);
    }
    return s;
}
```

In this example, `argv1` is converted to an integer with `atoi`. `atoi` does not provide errors for an invalid integer string. The conversion can fail unexpectedly.

#### Correction — Use `strtol` instead

One possible correction is to use `strtol` to validate the input string and the converted integer.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char *argv1)
{
    char *c_str = argv1;
    char *end;
    long sl;

    if (demo_check_string_not_empty(c_str))
    {
        errno = 0; /* set errno for error check */
        sl = strtol(c_str, &end, 10);
        if (end == c_str)
        {
            (void)fprintf(stderr, "%s: not a decimal number\n", c_str);
        }
        else if ('\0' != *end)
        {
            (void)fprintf(stderr, "%s: extra characters: %s\n", c_str, end);
        }
        else if ((LONG_MIN == sl || LONG_MAX == sl) && ERANGE == errno)
        {
            (void)fprintf(stderr, "%s out of range of type long\n", c_str);
        }
        else if (sl > INT_MAX)
        {
            (void)fprintf(stderr, "%ld greater than INT_MAX\n", sl);
        }
        else if (sl < INT_MIN)
        {
            (void)fprintf(stderr, "%ld less than INT_MIN\n", sl);
        }
        else
        {
            return (int)sl;
        }
    }
    return 0;
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off

**Command-Line Syntax:** `UNSAFE_STR_TO_NUMERIC`
**Impact:** Low

## Version History

**Introduced in R2016b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Use of indeterminate string

Use of unvalidated buffer from fgets-family function

## Description

This defect occurs when you do not check if a write operation using an `fgets`-family function such as:

```
char * fgets(char* buf, int n, FILE *stream)
```

succeeded and the buffer written has valid content, or you do not reset the buffer on failure. You then perform an operation that assumes a buffer with valid content. For instance, if the buffer with possibly indeterminate content is `buf` (as shown above), the checker raises a defect if:

- You pass `buf` as argument to standard functions that print or manipulate strings or wide strings.
- You return `buf` from a function.
- You pass `buf` as argument to external functions with parameter type `const char *` or `const wchar_t *`.
- You read `buf` as `buf[index]` or `*(buf + offset)`, where `index` or `offset` is a numerical value representing the distance from the beginning of the buffer.

### Risk

If an `fgets`-family function fails, the content of its output buffer is indeterminate. Use of such a buffer has undefined behavior and can result in a program that stops working or other security vulnerabilities.

### Fix

Reset the output buffer of an `fgets`-family function to a known string value when the function fails.

## Examples

### Output of `fgets()` Passed to External Function

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func(void) {
    char buf[SIZE20];

    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* 'buf' may contain an indeterminate string.  */
        ;
    }
    /* 'buf passed to external function */
    display_text(buf);
}
```

In this example, the output `buf` is passed to the external function `display_text()`, but its value is not reset if `fgets()` fails.

#### Correction — Reset `fgets()` Output on Failure

If `fgets()` fails, reset `buf` to a known value before you pass it to an external function.

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func1(void) {
    char buf[SIZE20];
    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* value of 'buf' reset after fgets() failure. */
        buf[0] = '\0';
    }
    /* 'buf' passed to external function */
    display_text(buf);
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `INDETERMINATE_STRING`
**Impact:** Medium

# Version History

**Introduced in R2017b**

## See Also

`Invalid use of standard library string routine` | `Returned value of a sensitive function not checked` | `Use of dangerous standard function` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"

“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Use of memset with size argument zero

Size argument of function in `memset` family is zero

## Description

This defect occurs when you call a function in the `memset` family with size argument zero. Functions include `memset`, `wmemset`, `bzero`, `SecureZeroMemory`, `RtlSecureZeroMemory`, and so on.

### Risk

`void *memset (void *ptr, int value, size_t num)` fills the first `num` bytes of the memory block that `ptr` points to with the specified `value`. A zero value of `num` renders the call to `memset` redundant. The memory that `ptr` points to:

- Remains uninitialized, if not previously initialized.
- Is not cleared and can contain sensitive data, if previously initialized.

### Fix

Determine if the zero size argument occurs because of a previous error in your code. Fix the error.

## Examples

### Zero Size Argument of memset

```
#include <stdio.h>
#include <string.h>

void func (unsigned int size)
{
    char str[] = "Buffer to be filled.";
    memset (str,'-',size);
    puts (str);
}

void calling_func(void) {
    unsigned int buf_size=0;
    func(buf_size);
}
```

In this example, the argument `size` of `memset` is zero.

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `MEMSET_INVALID_SIZE`
**Impact:** Medium

## Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)` | `Call to memset with unintended value`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Variable length array with non-positive size

Size of variable-length array is zero or negative

## Description

This defect occurs when size of a variable-length array is zero or negative.

### Risk

If the size of a variable-length array is zero or negative, unexpected behavior can occur, such as stack overflow.

### Fix

When you declare a variable-length array as a local variable in a function:

- If you use a function parameter as the array size, check that the parameter is positive.
- If you use the result of a computation on a function parameter as the array size, check that the result is positive.

You can place a test for positive value either before the function call or the array declaration in the function body.

## Examples

### Nonpositive Array Size

```
int input(void);

void add_scalar(int n, int m) {
    int r=0;
    int arr[m][n];
    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            arr[i][j] = input();
            r += arr[i][j];
        }
    }
}

void main() {
    add_scalar(2,2);
    add_scalar(-1,2);
    add_scalar(2,0);
}
```

In this example, the second and third calls to `add_scalar` result in a negative and zero size of `arr`.

#### Correction — Make Array Size Positive

One possible correction is fix or remove calls that result in a nonpositive array size.

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `NON_POSITIVE_VLA_SIZE`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Writing to const qualified object

Object declared with a `const` qualifier is modified

## Description

This defect occurs when you do one of the following:

- Use a `const`-qualified object as the destination of an assignment.
- Pass a `const`-qualified object to a function that modifies the argument.

For instance, the defect can occur in the following situations:

- You pass a `const`-qualified object as first argument of one of the following functions:
  - `mkstemp`
  - `mkostemp`
  - `mkostemps`
  - `mkdtemp`
- You pass a `const`-qualified object as the destination argument of one of the following functions:
  - `strcpy`
  - `strncpy`
  - `strcat`
  - `memset`
- You perform a write operation on a `const`-qualified object.

### Risk

The risk depends upon the modifications made to the `const`-qualified object.

| Situation | Risk |
|---|---|
| Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on. | These functions replace the last six characters of their first argument with a string. Therefore, they expect a modifiable `char` array as their first argument. |
| Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on. | These functions modify their destination argument. Therefore, they expect a modifiable `char` array as their destination argument. |
| Writing to the object | The `const` qualifier implies an agreement that the value of the object will not be modified. By writing to a `const`-qualified object, you break the agreement. The result of the operation is undefined. |

### Fix

The fix depends on the modification made to the `const`-qualified object.

| Situation | Fix |
|---|---|
| Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on. | Pass a non-`const` object as first argument of the function. |
| Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on. | Pass a non-`const` object as destination argument of the function. |
| Writing to the object | Perform the write operation on a non-`const` object. |

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Examples

### Writing to `const`-Qualified Object

```
#include <string.h>

const char* buffer = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

In this example, because `buffer` is `const`-qualified, `strchr(buffer,'X')` returns a `const`-qualified `char*` pointer. When this `char*` pointer is used as the destination argument of `strcpy`, a **Writing to const qualified object** error appears.

#### Correction — Copy `const`-Qualified Object to Non-`const` Object

One possible correction is to assign the constant string to a non-`const` object and use the non-`const` object as destination argument of `strchr`.

```
#include <string.h>

char buffer[] = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `CONSTANT_OBJECT_WRITE`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Wrong type used in sizeof

`sizeof` argument does not match pointed type

## Description

This defect occurs when both of the following conditions hold:

1. You assign the address of a block of memory to a pointer, or transfer data between two blocks of memory. The assignment or copy uses the `sizeof` operator.

   For instance, you initialize a pointer using `malloc(sizeof(`*`type`*`))` or copy data between two addresses using `memcpy(`*`destination_ptr`*`, `*`source_ptr`*`, sizeof(`*`type`*`))`.
2. You use an incorrect type as argument of the `sizeof` operator. For instance:

   - You might be using the pointer type instead of the type that the pointer points to. For example, to initialize a *`type`*`*` pointer, you might be using `malloc(sizeof(`*`type`*`*))` instead of `malloc(sizeof(`*`type`*`))`.
   - You might be using a completely unrelated type as `sizeof` argument. For example, to initialize a *`type`*`*` pointer, you might be using `malloc(sizeof(`*`anotherType`*`))`.

### Risk

Irrespective of what *`type`* stands for, the expression `sizeof(`*`type`*`*)` always returns a fixed size. The size returned is the pointer size on your platform in bytes. The appearance of `sizeof(`*`type`*`*)` often indicates an unintended usage. The error can cause allocation of a memory block that is much smaller than what you need and lead to weaknesses such as buffer overflows.

For instance, assume that `structType` is a structure with ten `int` variables. If you initialize a `structType*` pointer using `malloc(sizeof(structType*))` on a 32-bit platform, the pointer is assigned a memory block of four bytes. However, to be allocated completely for one `structType` variable, the `structType*` pointer must point to a memory block of `sizeof(structType) = 10 * sizeof(int)` bytes. The required size is much greater than the actual allocated size of four bytes.

### Fix

To initialize a *`type`*`*` pointer, replace `sizeof(`*`type`*`*)` in your pointer initialization expression with `sizeof(`*`type`*`)`.

## Examples

### Allocate a Char Array With `sizeof`

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char*) * 5);
    free(str);
}
```

In this example, memory is allocated for the character pointer `str` using a `malloc` of five char pointers. However, `str` is a pointer to a character, not a pointer to a character pointer. Therefore the `sizeof` argument, `char*`, is incorrect.

#### Correction — Match Pointer Type to `sizeof` Argument

One possible correction is to match the argument to the pointer type. In this example, `str` is a character pointer, therefore the argument must also be a character.

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char) * 5);
    free(str);

}
```

### Result Information

**Group:** Programming
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `PTR_SIZEOF_MISMATCH`
**Impact:** High

## Version History

**Introduced in R2013b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Non-compliance with AUTOSAR specification

An RTE API function is used with arguments that violate the AUTOSAR standard specification

## Description

This defect occurs when you use an RTE API function with arguments that violate the AUTOSAR standard specifications.

For instance, checks on `Rte_Write_*` or `Rte_Byps_Write_*` function calls determine if the pointer-to-data argument in the call:

- Is NULL valued.
- Points to a memory buffer.
- Points to an initialized memory buffer.
- For buffers with enum values, values are within the enum range.

For more information on the RTE API specifications, see AUTOSAR documentation (Specification of RTE Software).

To enable this check, use the value `autosar` for the option `Libraries used (-library)`.

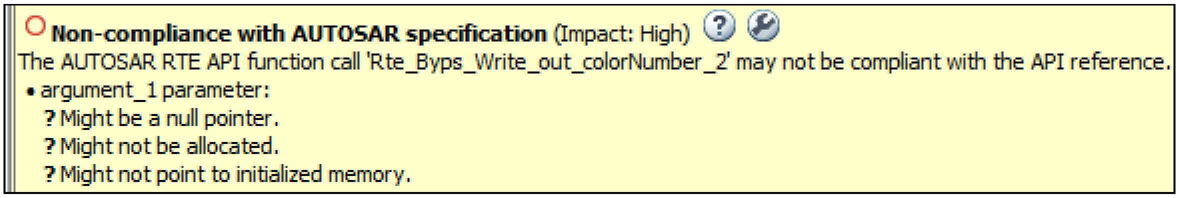
A more exhaustive version of the same checker is available with Code Prover. When checking for AUTOSAR standard violations on an `Rte_` function call, the Code Prover checker considers *all* execution paths that lead to the function call (subject to verification assumptions).

### Risk

The RTE function usage might lead to run-time errors.

### Fix

The fix depends on the root cause of the defect. To diagnose this check, read the message on the **Result Details** pane. The message shows all checks performed on the RTE API function, along with information about whether the check passed. For instance, this message:



○ **Non-compliance with AUTOSAR specification** (Impact: High)
The AUTOSAR RTE API function call 'Rte_Byps_Write_out_colorNumber_2' may not be compliant with the API reference.
- argument_1 parameter:
  **?** Might be a null pointer.
  **?** Might not be allocated.
  **?** Might not point to initialized memory.

Shows the results of three checks, all three of which might fail. The first argument of the function might be a null pointer, might not be allocated and might not point to initialized memory.

Investigate the root cause of the issue further.

## Examples

### Rte_Byps_Write_* Argument Pointing to Statically Allocated Noninitialized Buffer

```
#include <stdlib.h>

// Type declarations that are typically in AUTOSAR header Rte_type.h
typedef unsigned char uint8_T;
typedef unsigned int uint32_T;
typedef uint8_T Std_ReturnType;

typedef struct {
    uint8_T color;
    uint32_T number;
}
colorNumber;

extern Std_ReturnType Rte_Byps_Write_out_colorNumber_1(colorNumber*);

void SendData() {
    colorNumber aColor;
    uint8_T copyColor;
    uint32_T copyNumber;

    colorNumber* aPtrColor = &aColor;
    Rte_Byps_Write_out_colorNumber_1(aPtrColor);

    copyColor = aColor.color;
    copyNumber = aColor.number;
}
```

In this example, the function `Rte_Byps_Write_out_colorNumber_1` takes a pointer to a noninitialized variable. The checker flags the function call because the pointer does not point to initialized memory. To run this example, use the option `-library autosar`.

### Rte_Byps_Write_* Argument Pointing to Dynamically Allocated Noninitialized Buffer

```
#include <stdlib.h>

// Type declarations that are typically in AUTOSAR header Rte_type.h
typedef unsigned char uint8_T;
typedef unsigned int uint32_T;
typedef uint8_T Std_ReturnType;

typedef struct {
    uint8_T color;
    uint32_T number;
}
colorNumber;
extern Std_ReturnType Rte_Byps_Write_out_colorNumber_2(colorNumber*);

void SendData() {
    colorNumber* arrayColorNumber = (colorNumber*) malloc(2*sizeof(colorNumber));
    uint8_T copyColor;
    uint32_T copyNumber;
    Rte_Byps_Write_out_colorNumber_2(arrayColorNumber);

    copyColor = arrayColorNumber[0].color;
    copyNumber = arrayColorNumber[0].number;
}
```

In this example, the function `Rte_Byps_Write_out_colorNumber_2` takes a pointer returned from a memory allocation with `malloc`. The checker flags the function call because the pointer does not point to initialized memory. To run this example, use the option `-library autosar`.

## Result Information

**Group:** Programming
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `autosar_lib_non_compliance`
**Impact:** High

# Version History

**Introduced in R2021a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Result of `string::c_str()` compared to another pointer

The C string obtained from `std::string::c_str()` is compared to a pointer (or NULL)

## Description

This defect occurs when a C string that is obtained by calling the `std::string::c_str` function is compared to a pointer or a NULL. For instance, Polyspace flags the comparison operations in the `if` statements in this code:

```
void foo(){
    //...
    std::string str{"loren ipsum"};
    //...
    const char pStr[] = "loren ipsum";
    const char* p = str.c_str();
    if(p==NULL){//Defect: Unnecessary

    }
    if(p==pStr){ //Defect: Compares pointer address
    //..
    }
    //..
}
```

### Risk

Comparing a pointer to the C string obtained from a `string` has these risks:

- When you compare the output of `std::string::c_str` to a pointer, the addresses of the pointers are compared. You might expect the compiler to compare the content of the pointers. For instance, in the preceding code, you might expect that `(p==pStr)` evaluates to `true` because both pointers contains `loren ipsum`. The compiler compares the addresses `p` and `pStr`, which evaluates to `false`. Comparing pointers as a method of comparing strings produces unexpected results.
- The C string `p` that is obtained by calling `std::string::c_str()` is always non-NULL. The expression `(p==NULL)` always evaluates to `false`. Comparing such a C string to NULL might produce unexpected results and indicates a logic error in the code.

### Fix

To fix this issue:

- To compare the content of strings, use string functions or use operators with the string objects directly.
- Because `std::string::c_str()` always returns a non-NULL value, remove the comparison to NULL or refactor your logic.

## Examples

### Avoid Comparing C Strings from string::c_str() to NULL

```
#include<string>
extern void setCategoryName(const char *);

void setName(const std::string &s)
{
    if (s.c_str() != NULL)//Defect
    {
        setCategoryName(s.c_str());
    }
}
```

In this example, the function setName sets the category name when the C string obtained from the string s is non-NULL. You might expect the condition (s.c_str() != NULL) to evaluate to true only when s contains a string of nonzero length. Because std::string::c_str() returns a non-NULL value regardless of the content of the string object, the condition always evaluates to true. As a result, this comparison is unnecessary and cannot detect empty strings.

#### Correction — Use string::empty to Detect Empty

To fix this defect, remove the comparison of the C string to NULL. To detect empty string, use std::string::empty().

```
#include<string>

extern void setCategoryName(const char *);
void setName_possibleFix1(const std::string &s)
{
    setCategoryName(s.c_str());
}

void setName_possibleFix2(const std::string &s)
{
    if (!s.empty())
    {
        setCategoryName(s.c_str());
    }
}
```

### Avoid Comparing C Strings From string::c_str() to Other Pointers

```
#include<string>
bool is_same(const std::string &s1, const std::string &s2)
{
    const char* pStr1 = s1.c_str();
    const char* pStr2 = s2.c_str();
    return (pStr1 == pStr2);//Defect
}
bool is_different(const std::string &s1, const char* Cstr)
{
    const char* pStr1 = s1.c_str();
    return (pStr1 != Cstr);//Defect
}
```

In this example, C strings pStr1 and pStr2 are obtained by calling string::c_str() and then compared to other pointers. In is_same, you might expect (pStr1 == pStr2) to evaluate to true if the contents of s1 and s2 contains the same string. Because this operation is a pointer comparison, it compares the address of the pointers pStr1 and pStr2 instead of the content of the string, producing unexpected results.

When comparing string objects and C strings, you might expects comparison operations such as (pStr1 != Cstr) to compare the content of the string and char pointer. Because this operation compares the addresses of the pointers instead of the string contents, the results might be unexpected.

#### Correction — Use Member Functions and Overloaded Operators of std::string

To fix this defect, use the member functions and overloaded operators of std::string to compare strings and C strings.

```
#include <string>

bool is_same(const std::string &s1, const std::string &s2)
{
    return (s1 == s2);
}
bool is_different(const std::string &s1, const char* Cstr)
{
    return (s1!= Cstr);
}
```

## Result Information

**Group:** Programming
**Language:** C++
**Default:** Off
**Command-Line Syntax:** STD_STRING_C_STR_COMPARED_TO_POINTER
**Impact:** Low

# Version History

**Introduced in R2021b**

## See Also

Find defects (-checkers)

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Improper erase-remove idiom

Container's `erase()` is not called or called improperly following a call to `std::remove()`

## Description

This defect occurs when any of these conditions are true:

- The function `std::remove()` or `std::remove_if()` is called and the output is not passed to a container's `erase()` method.
- The container's `erase()` method is called by using the output of `std::remove` but without passing the second parameter.

### Risk

You might expect `std::remove()` or `std::remove_if` to remove entries from a container. Instead, these functions partition a container and move the entries that match the removal criteria to the right side. This right side content might not be meaningful. If you do not pass the output of `std::remove()` and `std::remove_if` to the container's `erase()` method, the entries are not removed and the container might remain in an indeterminate state.

When you call a container's `erase()` by using the output of `std::remove` but do not specify the second parameter, the call to `erase()` might lead to unexpected or undefined behavior. For instance, such a call to `erase()` removes only the first entry on the right side of the container instead of removing all of them. If `std::remove()` returns the `end()` iterator, such a malformed `erase()` results in undefined behavior.

### Fix

To remove entries from a container, use these two steps:

1. First, call `std::remove()` or `std::remove_if` to move the entries that you want to remove to the right side of the container.
2. Then, call the container's `erase()` method to remove the entries on the right side of the container.

Avoid using `std::remove()` without a following call to `erase()`. Some containers implement their own `remove()` methods. For these containers, it might be sufficient to use these `remove()` methods instead of `std::remove()`.

If you want to partition a container, use `std::partition()`.

If you use C++20, use the functions `std:erase()` and `std::erase_if()`. These functions implement the erase-remove idiom in a single call.

When calling the containers `erase()` with the output of `std::remove()`, specify the second parameter of `erase()` to avoid undefined behaviors. Generally, the second parameter of `erase()` matches the second parameter of `std::remove()`.

## Examples

### Use Erase-Remove Idiom to Remove Entries from Containers Efficiently

```
#include <vector>
#include<string>
#include <algorithm>
typedef std::string V;

void removeValues(std::vector<V> &vec, const V &value)
{
    std::remove(vec.begin(), vec.end(), value);
}
void removeValues_improper(std::vector<V> &vec, const V &value)
{
    vec.erase(std::remove(vec.begin(), vec.end(), value));
}
```

In this example, Polyspace flags improper use of the erase-remove idiom.

- In the function `removeValues`, Polyspace flags the use of `std::remove` because its output is not used. Because the output is not passed to `vec.erase()`, the entries are not erased from the vector and the vector remains in an indeterminate state.
- In the function `removeValues_improper`, Polyspace flags the improper call to `vec.erase()`. Because the call has no second parameter, `vec.erase()` might erase only the first element of the range that is returned by `std::remove()`. If `std::remove()` returns the `end()` iterator, this call to `vec.erase()` is undefined behavior.

#### Correction — Proper implementation of erase-remove idiom

To fix this defect, pass the output of `std::remove()` to `vec.erase()`. Specify a second parameter when calling `vec.erase()`. The second parameter for `vec.erase()` might be the same as the second parameter to `std::remove()`.

```
#include <vector>
#include<string>
#include <algorithm>
typedef std::string V;
void removeValues_FIXED(std::vector<V> &vec, const V &value)
{
    vec.erase(std::remove(vec.begin(), vec.end(), value), vec.end());
}
```

## Result Information

**Group:** Programming
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `STD_REMOVE_WITHOUT_ERASE`
**Impact:** Medium

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Invalid iterator usage

Mismatched or uninitialized iterators are used in standard algorithm functions and comparison operations

## Description

This checker flags invalid use of iterators. For instance, this checker is raised when:

- Iterators from mismatched containers are compared to each other.
- Iterators from mismatched containers are passed to algorithms that expect iterators from the same container.
- Uninitialized iterators are passed to a standard algorithm function or used in a comparison operation.
- The `end()` iterator is passed as the only argument to the `vector::erase()` method.

### Risk

The preceding invalid uses of iterators might result in unexpected behavior.

- Calling standard algorithms by using mismatched or uninitialized iterators might lead to segmentation faults or undefined behaviors.
- Because mismatched iterators point to different containers, comparing them fails silently. This behavior might be unexpected and lead to bugs that are difficult to diagnose.

### Fix

The preceding invalid use of iterators might indicate typographical errors or logic errors in your code. To fix such defects, use initialized iterators from the same container in standard algorithms and comparison operations.

## Examples

### Mismatched Iterators in Standard Algorithm Function

```
#include <vector>
#include <algorithm>
#include <string>
typedef const std::vector<std::string>  VS;

bool bothHaveKey(VS& v1, VS& v2, const std::string& key)
{
    bool b1 = (std::find(v1.begin(), v2.end(), key) != v1.end());

    bool b2 = (std::find(v2.begin(), v1.end(), key) != v2.end());
    return b1 && b2;
}
```

In this example, the function `std::find` expects a first/last pair of iterators from the same container. Because this function is called by using mismatched iterators from different containers `v1` and `v2`, Polyspace flags these function calls.

#### Correction — Use Iterators from Same Container

To fix this defect, use iterators that point to the same container when calling `std::find`.

```
#include <vector>
#include <algorithm>
#include <string>
typedef const std::vector<std::string> VS;

bool bothHaveKey(VS& v1, VS& v2, const std::string& key)
{
    bool b1 = (std::find(v1.begin(), v1.end(), key) != v1.end());

    bool b2 = (std::find(v2.begin(), v2.end(), key) != v2.end());

    return b1 && b2;
}
```

### Uninitialized Iterators in Standard Algorithm Function

```
#include <vector>
#include <algorithm>
#include <string>
typedef std::vector<std::string>::iterator vs_it;
vs_it first;
vs_it last;

//first and last are uninitialized
auto findBetween(std::string key)
{
    return std::find(first, last, key); //Noncompliant
}
```

In this example, the `std::find` method is called by using uninitialized iterators `first` and `last`. Such a call might lead to undefined behaviors. Polyspace flags the call.

#### Correction — Initialize Iterators Before Use

To fix this defect, initialize the iterators before calling `std::find`.

```
#include <vector>
#include <algorithm>
#include <string>

typedef std::vector<std::string>::iterator vs_it;
std::vector<std::string> V;

vs_it first = V.begin();
vs_it last = V.end();

auto findBetween(std::string key)
{
    return std::find(first, last, key);
}
```

### Comparing Mismatched Iterators

```
#include <vector>
#include <algorithm>
#include <string>
typedef std::vector<std::string> VS;

std::string key;

bool foo(VS v1, VS v2)
{
    return std::find(v1.begin(), v1.end(), key) == v2.end();
}
```

In this example, iterators from the containers `v1` and `v2` are compared in the function `foo`. This operation always returns false, which might be unexpected. Polyspace flags the operation.

#### Correction — Compare Iterators from Same Container

To fix this defect, compare iterators from the same container.

```
#include <vector>
#include <algorithm>
#include <string>
typedef std::vector<std::string> VS;

std::string key;

bool foo(VS v1, VS v2)
{
    return std::find(v1.begin(), v1.end(), key) == v1.end();
}
```

## Result Information

**Group:** Programming
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `INVALID_ITERATOR_USAGE`
**Impact:** High

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"

"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# std::string_view initialized with dangling pointer

An `std::string_view` object is initialized by using an unnamed temporary object

## Description

This defect occurs when you construct an `std::string_view` object by using an unnamed temporary object.

```
std::string get();
std::string_view s = get(); // Defect
```

In the preceding code, `get()` returns a temporary string that goes out of scope after the semicolon. The `string_view` object `s` remains in scope, but it points to memory that is already released. Polyspace flags initializing `string_view` objects with such dangling pointers.

This defect is relevant for C++17 or later.

### Risk

An `std::string_view` object typically contains a pointer to a `const char` array. When you initialize an `std::string_view` object by using an unnamed temporary, the pointer in the `std::string_view` object points to the memory block containing the temporary. The unnamed temporary goes out of scope at the end of the statement where it is created, and releases the memory. The `std::string_view` object then contains a dangling pointer, which leads to bugs that are difficult to find.

### Fix

Avoid using unnamed temporaries when initializing `std::string_view` objects. Use named variables instead.

## Examples

### Avoid Initializing `std::string_view` Objects by Using Unnamed Temporaries

```
#include <string_view>
#include <string>

extern std::string getString();

extern void useStringView(std::string_view s);

void foo()
{
    std::string_view sv = getString();

    useStringView(sv);
}
```

In this example, the `std::string_view` object `sv` is initialized by the temporary object returned by `getString`. The temporary goes out of scope at the end of the initialization statement. The function `useStringView()` is called with a dangling `std::string_view`. This call might result in unexpected behavior. Polyspace raises a defect on the initialization statement.

#### Correction — Use Named Variables When Initializing `std::string_view`

To fix this defect, initialize `std::string_view` objects by using named variables.

```
#include <string_view>
#include <string>

extern std::string getString();

extern void useStringView(std::string_view s);

void foo()
{
    const std::string s = getString();
    std::string_view sv = s;
    useStringView(sv);
}
```

## Result Information

**Group:** Programming
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `DANGLING_STRING_VIEW`
**Impact:** High

# Version History

**Introduced in R2022b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

12

# Data Flow Defects

# Code deactivated by constant false condition

Code segment deactivated by `#if 0` directive or `if(0)` condition

## Description

This defect occurs when a block of code is deactivated using a `#if 0` directive or `if(0)` condition.

### Risk

A `#if 0` directive or `if(0)` condition is used to temporarily deactivate segments of code. If your production code contains these directives, it means that the deactivation has not been lifted before shipping the code.

### Fix

If the segment of code is present for debugging purposes only, remove the segment from production code. If the deactivation occurred by accident, remove the `#if 0` and `#endif` statements.

Often, a segment of code is deactivated for specific conditions, for instance, a specific operating system. Use macros with the `#if` directive to indicate these conditions instead of deactivating the code completely with a `#if 0` directive. For instance, GCC provides macros to detect the Windows operating system:

```
#ifdef _WIN32
   //Code deactivated for all operating systems
   //Other than 32-bit Windows
#endif
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Code Deactivated by Constant False Condition Error

```
#include<stdio.h>
int Trim_Value(int* Arr,int Size,int Cutoff)
{
    int Count=0;

    for(int i=0;i < Size;i++){
        if(Arr[i]>Cutoff){
            Arr[i]=Cutoff;
            Count++;
        }
    }

    #if 0
    /* Defect: Code Segment Deactivated */

    if(Count==0){
        printf("Values less than cutoff.");
    }
     #endif

    return Count;
}
```

In the preceding code, the `printf` statement is placed within a `#if #endif` directive. The software treats the portion within the directive as code comments and not compiled.

**Correction — Change `#if 0` to `#if 1`**

Unless you intended to deactivate the `printf` statement, one possible correction is to reactivate the block of code in the `#if #endif` directive. To reactivate the block, change `#if 0` to `#if 1`.

```
#include<stdio.h>
int Trim_Value(int* Arr,int Size,int Cutoff)
{
 int Count=0;

 for(int i=0;i < Size;i++)
     {
      if(Arr[i]>Cutoff)
            {
             Arr[i]=Cutoff;
             Count++;
            }
     }


 /* Fix: Replace #if 0 by #if 1 */
 #if 1
      if(Count==0)
           {
            printf("Values less than cutoff.");
           }
 #endif

 return Count;
}
```

## Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** off
**Command-Line Syntax:** `DEACTIVATED_CODE`
**Impact:** Low

## Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Unreachable code` | `Useless if` | `Dead code`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Dead code

Code does not execute

## Description

This defect occurs when a block of code cannot be reached because of a condition that is always true or false. This defect excludes:

- `Code deactivated by constant false condition`, which checks for directives with compile-time constants such as `#if 0` or `if(0)`.
- `Unreachable code`, which checks for code after a control escape such as `goto`, `break`, or `return`.
- `Useless if`, which checks for if statements that are always true.

### Risk

Dead code wastes development time, memory and execution cycles. Developers have to maintain code that is not being executed. Instructions that are not executed still have to be stored and cached.

Dead code often represents legacy code that is no longer used. Cleaning up dead code periodically reduces future maintenance.

### Fix

The fix depends on the root cause of the defect. For instance, the root cause can be an error condition that is checked twice on the same execution path, making the second check redundant and the corresponding block dead code.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you see dead code from use of functions such as `isinf` and `isnan`, enable an analysis mode that takes into account non-finite values. See `Consider non finite floats (-allow-non-finite-floats)`.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Examples

### Dead Code from if-Statement

```
#include <stdio.h>

int Return_From_Table(int ch){

    int table[5];

    /* Create a table */
    for(int i=0;i<=4;i++){
        table[i]=i^2+i+1;
    }

    if(table[ch]>100){ /* Defect: Condition always false */
         return 0;
    }
    return table[ch];
}
```

The maximum value in the array `table` is `4^2+4+1=21`, so the test expression `table[ch]>100` always evaluates to false. The `return 0` in the `if` statement is not executed.

#### Correction — Remove Dead Code

One possible correction is to remove the `if` condition from the code.

```
#include <stdio.h>

int Return_From_Table(int ch){

    int table[5];

    /* Create a table */
    for(int i=0;i<=4;i++){
        table[i]=i^2+i+1;
    }

    return table[ch];
}
```

### Dead Code for if with Enumerated Type

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;

    if (card > 7) {
        do_something(card);
    }
}
```

The type `suit` is enumerated with five options. However, the conditional expression `card > 7` always evaluates to false because `card` can be at most 5. The content in the `if` statement is not executed.

#### Correction — Change Condition

One possible correction is to change the if-condition in the code. In this correction, the 7 is changed to `HEART` to relate directly to the type of `card`.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;

    if (card > HEARTS) {
        do_something(card);
    }
}
```

### Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `DEAD_CODE`
**Impact:** Low

## Version History

**Introduced in R2013b**

### See Also

`Find defects (-checkers)` | `Unreachable code` | `Useless if` | `Code deactivated by constant false condition`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Infinite loop

Loop termination condition might never be satisfied

## Description

This defect occurs when a loop termination condition is never satisfied after the loop is entered.

The checker skips intentional infinite loops such as `while(1)`. Code following intentional infinite loops are flagged by the `Unreachable code` checker.

### Risk

Unintended infinite loops often indicate an error in the program logic. For instance, one of the following programming errors might have occurred:

- The loop index is never updated inside the loop.
- The loop index is updated in branches inside the loop that are unreachable.
- The loop index is updated in a way that the index never satisfies the loop termination condition.

### Fix

Fix the programming error if any. Make sure that the loop index update is reachable, and the loop index eventually acquires a value that makes the loop terminate.

If you determine that the loop termination condition is satisfied under certain circumstances (false positive), add comments to your result or code to avoid another review. See

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Loop Index Not Updated

```
void cleanArray(int* fullArray, int* finalArray, int lenFullArray, int lenfinalArray) {
    int i,j;
    for(i = 0, j = 0; i < lenFullArray; j++) {
        if(fullArray[i] >= 0 && j < lenfinalArray) {
            finalArray[j] = fullArray[i];
            j++;
        }
    }
}
```

In this example, the loop uses two indices to cycle through two arrays. The index used in the loop termination condition is never updated inside the loop, possibly because of a programming error (the other index is updated twice).

**Correction – Update Loop Index**

Make sure to update the loop index used in the loop termination condition.

```
void cleanArray(int* fullArray, int* finalArray, int lenFullArray, int lenfinalArray) {
    int i,j;
    for(i = 0, j = 0; i < lenFullArray; i++) {
        if(fullArray[i] >= 0 && j < lenfinalArray) {
            finalArray[j] = fullArray[i];
            j++;
        }
    }
}
```

### Loop Index Not Updated in Code Under Analysis

```
#include <stdint.h>

uint32_t idx;

void func(void);

void foo() {
    while(idx) {
        func();
    }
}
```

In this example, the variable `idx` is seemingly not updated in the `while` loop, leading to the detection of an infinite loop.

The issue happens because the definition of the function `func()` is not available to the Bug Finder analysis. Therefore, the analysis stubs the function and assumes that the function does not update global variables, including the variable `idx`.

### Loop Index Updated in Unreachable Branch

```
int addNTimes(int elem, int addNegative, int addPositive, int init, int final) {
    int i = init, sum = elem;
    while (i <= final) {
        if(i < 0) {
            sum += addNegative;
            i++;
        }
        else {
            sum += addPositive;
        }
    }
    return sum;
}
```

In this example, if the fourth argument `init` and the fifth argument `final` are both positive, the loop index is never updated. The loop index update occurs in the branch `if(i < 0)`, which is never reached.

Note that Bug Finder can find the issue only if you analyze the code with the fourth and fifth arguments constrained to positive values. In other words, specify the following constraints on `arg4` and `arg5`:

- **Init Mode**: `INIT`
- **Init Range**: `0..max`

See also “Specify External Constraints for Polyspace Analysis” and “External Constraints for Polyspace Analysis”.

#### Correction – Update Loop Index Outside Branches

Unless specifically needed, update loop indices outside branches of a condition. You reduce the chances of a loop being infinite because the index was not updated due to conditional logic.

```
int addNTimes(int elem, int addNegative, int addPositive, int init, int final) {
    int i = init, sum = elem;
    while (i <= final) {
        if(i < 0) {
            sum += addNegative;
        }
        else {
            sum += addPositive;
        }
        i++;
    }
    return sum;
}
```

## Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `INFINITE_LOOP`
**Impact:** High

# Version History

**Introduced in R2023a**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Missing return statement

Function with non-`void` return type does not return value on some paths

## Description

This defect occurs when a function does not return a value along at least one execution path. This defect does not occur if:

- The return type of the function is `void`.
- The execution path is terminated by a function that does not return the flow of execution, such as a `[[noreturn]]` function.
- If you use C version above C99 or C++. From C version C99 and above, as well as for C++, the `main` function implicitly returns a `0` if a return value is not specified.

### Risk

If a function has a non-`void` return value in its signature, it is expected to return a value. The return value of this function can be used in later computations. If the execution of the function body goes through a path where a `return` statement is missing, the function return value is indeterminate. Computations with this return value can lead to unpredictable results.

### Fix

In most cases, you can fix this defect by placing the `return` statement at the end of the function body. If your code has execution paths that do not return the flow of execution, specify them by using the attribute `[[noreturn]]`.

Alternatively, you can identify which execution paths through the function body do not have a `return` statement and add a `return` statement on those paths. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

If the analysis flags a missing `return` statement on a path where a process termination function exists, you can make the analysis aware of the process termination function using the option `-termination-functions`.

## Examples

### Missing or invalid return statement error

```
int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }
 }
/* Defect: No return value if n is not 0*/
```

If n is equal to 0, the code does not enter the if statement. Therefore, the function AddSquares does not return a value if n is 0.

#### Correction — Place Return Statement on Every Execution Path

One possible correction is to return a value in every branch of the if...else statement.

```
 int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }

   /*Fix: Place a return statement on branches of if-else */
   else
     return 0;
  }
```

## Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** MISSING_RETURN
**Impact:** Low

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Non-initialized pointer

Pointer not initialized before dereference

## Description

This defect occurs when a pointer is not assigned an address before dereference.

### Risk

Unless a pointer is explicitly assigned an address, it points to an unpredictable location.

### Fix

The fix depends on the root cause of the defect. For instance, you assigned an address to the pointer but the assignment is unreachable.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below. It is a good practice to initialize a pointer to NULL when declaring the pointer.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

If a pointer in your code is non-initialized only for certain system input values, you can see one possible combination of input values causing the defect. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Non-initialized pointer error

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;
    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j;
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

**Correction — Initialize Pointer on Every Execution Path**

One possible correction is to assign an address to `pi` when `prev` is not `NULL`.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
       {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
       }
    /* Fix: Initialize pi in branches of if statement  */
    else
        pi = prev;


    *pi = j;

    return pi;
}
```

## Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `NON_INIT_PTR`
**Impact:** High

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Non-initialized variable`

**Topics**

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Non-initialized variable

Variable not initialized before use

## Description

This defect occurs when a variable is not initialized before its value is read.

### Risk

Unless a variable is explicitly initialized, the variable value is unpredictable. You cannot rely on the variable having a specific value.

### Fix

The fix depends on the root cause of the defect. For instance, you assigned a value to the variable but the assignment is unreachable or you assigned a value to the variable in one of two branches of a conditional statement. Fix the unreachable code or missing assignment.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below. It is a good practice to initialize a variable at declaration.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

You can extend the checker in the following ways:

- Polyspace does not flag passing pointers to noninitialized variables to functions. To detect noninitialized variables that are passed to functions by pointers, extend the checker by using the option `-code-behavior-specification`. See “Extend Checkers for Initialization to Check Function Arguments Passed by Pointers”.
- If a variable in your code is non-initialized only for certain system input values, you can see one possible combination of input values causing the defect. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
    /* Defect: val does not have a value if command is not 2 */
}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

#### Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that the initialization is not bypassed on some execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

## Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `NON_INIT_VAR`
**Impact:** High

## Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Non-initialized pointer`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Checkers for Initialization to Check Function Arguments Passed by Pointers"

# Partially accessed array

Array partly read or written before end of scope

## Description

This defect occurs when an array is partially read or written before the end of array scope. For arrays local to a function, the end of scope occurs when the function ends.

### Risk

A partially accessed array often indicates an omission in coding. For instance, when sorting an array using a loop, you used a number of loop iterations such that one array element is never read. The implementation can result in an array that is not fully sorted.

### Fix

The fix depends on the root cause of the defect. For instance, if the root cause is a loop with an incorrect number of iterations, change the loop bound or add a step after the loop to access the unread or unwritten elements.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Partially accessed array error

```
int Calc_Sum(void)
{
  int tab[5]={0,1,2,3,4},sum=0;
  /* Defect: tab[4] is not read */

  for (int i=0; i<4;i++) sum+=tab[i];

  return(sum);

 }
```

The array `tab` is only partially read before end of function `Calc_Sum`. While calculating `sum`, `tab[4]` is not included.

#### Correction — Access Every Array Element

One possible correction is to read every element in the array `tab`.

```
int Calc_Sum(void)
{
  int tab[5]={0,1,2,3,4},sum=0;

  /* Fix: Include tab[4] in calculating sum */
  for (int i=0; i<5;i++) sum+=tab[i];

  return(sum);

 }
```

## Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `PARTIALLY_ACCESSED_ARRAY`
**Impact:** Low

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Pointer to non initialized value converted to const pointer

Pointer to constant assigned address that does not contain a value

## Description

This defect occurs when a pointer to a constant (`const int*`, `const char*`, etc.) is assigned an address that does not yet contain a value.

For instance:

```
int x; const int * ptr = &x;
```

### Risk

A pointer to a constant stores a value that must not be changed later in the program. If you assign the address of a non-initialized variable to the pointer, it now points to an address with garbage values for the remainder of the program.

### Fix

Initialize a variable before assigning its address to a pointer to a constant.

### Extend Checker

If a `const` pointer is initialized incorrectly only for certain system input values, you can see one possible combination of input values causing the defect. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Address of Noninitialized Variable Assigned to `const` Pointer

```
#include<stdio.h>

void Display_Parity()
 {
  int num,parity;
  const int* num_ptr = &num;
  /* Defect: Address &num does not store a value */

  printf("Enter a number\n:");
  scanf("%d",&num);

  parity=((*num_ptr)%2);
  if(parity==0)
    printf("The number is even.");
  else
    printf("The number is odd.");
 }
```

num_ptr is declared as a pointer to a constant. However the variable num does not contain a value when num_ptr is assigned the address &num.

#### Correction — Initialize Variable Before Assigning Its Address to const Pointer

One possible correction is to obtain the value of num from the user before &num is assigned to num_ptr.

```
#include<stdio.h>

void Display_Parity()
 {
  int num,parity;
  const int* num_ptr;

  printf("Enter a number\n:");
  scanf("%d",&num);

 /* Fix: Assign &num to pointer after it receives a value */
  num_ptr=&num;
  parity=((*num_ptr)%2);
  if(parity==0)
    printf("The number is even.");
  else
    printf("The number is odd.");
 }
```

The scanf statement stores a value in &num. Once the value is stored, it is legitimate to assign &num to num_ptr.

### Address of Dynamically Allocated Memory Assigned to const Pointer

```
#include <stdlib.h>

int isElementInArray(const int* arr, const int elem);
void fillArray(int *);

void createArray(int elem) {
    int* arr = (int*) malloc (100*sizeof(int));
    isElementInArray(arr, elem);
}
```

In this example, the function isElementInArray takes a pointer to a const variable as first argument. The const specifier indicates that the function intends to read and not modify the pointed values. However, the array passed to this function is not initialized. The function isElementInArray, which can only read the array elements, reads non-initialized elements.

#### Correction – Initialize Memory Before Assigning to const Pointer

Initialize dynamically allocated memory before assigning its address to a const pointer. In this example, the array initialization is done using the fillArray function.

```
#include <stdlib.h>
```

```
int isElementInArray(const int* arr, const int elem);
void fillArray(int *);

void createArray(int elem) {
    int* arr = (int*) malloc (100*sizeof(int));
    fillArray(arr);
    isElementInArray(arr, elem);
}
```

## Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `NON_INIT_PTR_CONV`
**Impact:** Medium

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Static uncalled function

Function with static scope not called in file

## Description

This defect occurs when a `static` function is not called in the same file where it is defined.

### Risk

Uncalled functions often result from legacy code and cause unnecessary maintenance.

### Fix

If the function is not meant to be called, remove the function. If the function is meant for debugging purposes only, wrap the function definition in a debug macro.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Uncalled function error

Save the following code in the file `Initialize_Value.c`

```
#include <stdlib.h>
#include <stdio.h>

static int Initialize(void)
/* Defect: Function not called */
  {
   int input;
   printf("Enter an integer:");
   scanf("%d",&input);
   return(input);
  }

 void main()
  {
   int num;

   num=0;
   printf("The value of num is %d",num);
  }
```

The `static` function `Initialize` is not called in the file `Initialize_Value.c`.

**Correction — Call Function at Least Once**

One possible correction is to call `Initialize` at least once in the file `Initialize_Value.c`.

```
#include <stdlib.h>
#include <stdio.h>

static int Initialize(void)
  {
   int input;
   printf("Enter an integer:");
   scanf("%d",&input);
   return(input);
  }

 void main()
  {
   int num;

   /* Fix: Call static function Initialize */
   num=Initialize();

   printf("The value of num is %d",num);
  }
```

## Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `UNCALLED_FUNC`
**Impact:** Low

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unreachable code

Code not executed because of preceding control-flow statements

## Description

This defect occurs when a section of code cannot be reached because of a previous break in control flow.

Statements such as `break`, `goto`, and `return`, move the flow of the program to another section or function. Because of this flow escape, the statements following the control-flow code, statistically, do not execute, and therefore the statements are unreachable.

This check also finds code following trivial infinite loops, such as `while(1)`. These types of loops only release the flow of the program by exiting the program. This type of exit causes code after the infinite loop to be unreachable.

### Risk

Unreachable code wastes development time, memory and execution cycles. Developers have to maintain code that is not being executed. Instructions that are not executed still have to be stored and cached.

### Fix

The fix depends on the intended functionality of the unreachable code. If you want the code to be executed, check the placement of the code or the prior statement that diverts the control flow. For instance, if the unreachable code follows a `return` statement, you might have to switch their order or remove the `return` statement altogether.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Unreachable Code After Return

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if( (card < SPADES) || (card > CLUBS) )
        card = UNKNOWN_SUIT;
        return card;
    if (card < HEARTS) {
        guess(card);
    }
    return card;
}
```

In this example, there are missing braces and misleading indentation. The first `return` statement changes the flow of code back to where the function was called. Because of this return statement, the `if`-block and second `return` statement do not execute.

If you correct the indentation and the braces, the error becomes clearer.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if( (card < SPADES) || (card > CLUBS) ){
        card = UNKNOWN_SUIT;
    }
    return card;

    if (card < HEARTS) {
        guess(card);
    }
    return card;
}
```

**Correction — Remove Return**

One possible correction is to remove the escape statement. In this example, remove the first `return` statement to reach the final `if` statement.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if( (card < SPADES) || (card > CLUBS) )
    {
        card = UNKNOWN_SUIT;
    }

    if(card < HEARTS)
    {
        guess(card);
    }
    return card;
}
```

**Correction — Remove Unreachable Code**

Another possible correction is to remove the unreachable code if you do not need it. Because the function does not reach the second `if`-statement, removing it simplifies the code and does not change the program behavior.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if( (card < SPADES) || (card > CLUBS) )
    {
        card = UNKNOWN_SUIT;
    }
    return card;
}
```

### Infinite Loop Causing Unreachable Code

```
int add_apples(int apple) {
    int count = 1;
    while(1) {
        if(apple < 99){
            apple++;
            count++;
        }else{
            count--;
        }
    }
    return count;
}
```

In this example, the `while(1)` statement creates an infinite loop. The `return count` statement following this infinite loop is unreachable because the only way to exit this infinite loop is to exit the program.

#### Correction — Rewrite Loop Condition

One possible correction is to change the loop condition to make the `while` loop finite. In the example correction here, the loop uses the statement from the `if` condition: `apple < 99`.

```
int add_apples1(int apple) {
    int count = 0;
    while(apple < 99) {
        apple++;
        count++;
    }
    if(count == 0)
        count = -1;
    return count;
}
```

#### Correction — Add a Break Statement

Another possible correction is to add a break from the infinite loop, so there is a possibility of reaching code after the infinite loop. In this example, a `break` is added to the `else` block making the `return count` statement reachable.

```
int add_apples(int apple) {
    int count = 1;
    while(1) {
        if(apple < 99)
        {
            apple++;
            count++;
        }else{
            count--;
            break;
        }
    }
    return count;
}
```

#### Correction — Remove Unreachable Code

Another possible correction is to remove the unreachable code. This correction cleans up the code and makes it easier to review and maintain. In this example, remove the return statement and change the function return type to `void`.

```
void add_apples(int apple) {
    int count = 1;
    while(1) {
        if(apple < 99)
        {
            apple++;
            count++;
        }else{
            count--;
        }
    }
}
```

### Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `UNREACHABLE`
**Impact:** Medium

## Version History

**Introduced in R2013b**

### See Also

`Find defects (-checkers)` | `Dead code` | `Useless if` | `Code deactivated by constant false condition`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Useless if

Unnecessary if conditional

## Description

This defect occurs on `if`-statements where the condition is always true. This defect occurs only on if-statements that do not have an else-statement.

This defect shows unnecessary `if`-statements when there is no difference in code execution if the `if`-statement is removed.

### Risk

Unnecessary `if` statements often indicate a coding error. Perhaps the `if` condition is coded incorrectly or the `if` statement is not required at all.

### Fix

The fix depends on the root cause of the defect. For instance, the root cause can be an error condition that is checked twice on the same execution path, making the second check redundant.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If the redundant condition represents defensive coding practices and you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Examples

### `if` with Enumerated Type

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card < 7) {
        do_something(card);
    }
}
```

The type `suit` is enumerated with five options. However, the conditional expression `card < 7` always evaluates to true because `card` can be at most 5. The `if` statement is unnecessary.

**Correction — Change Condition**

One possible correction is to change the if-condition in the code. In this correction, the 7 is changed to `UNKNOWN_SUIT` to relate directly to the type of `card`.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card > UNKNOWN_SUIT) {
        do_something(card);
    }
}
```

**Correction — Remove If**

Another possible correction is to remove the if-condition in the code. Because the condition is always true, you can remove the condition to simplify your code.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    do_something(card);
}
```

## Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `USELESS_IF`

**Impact:** Medium

## Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)` | `Unreachable code` | `Dead code` | `Code deactivated by constant false condition`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Variable shadowing

Variable hides another variable of same name with nested scope

## Description

This defect occurs when a variable hides another variable of the same name in an outer scope.

For instance, if a local variable has the same name as a global variable, the local variable hides the global variable during its lifetime.

### Risk

When two variables with the same name exist in an inner and outer scope, any reference to the variable name uses the variable in the inner scope. However, a developer or reviewer might incorrectly expect that the variable in the outer scope was used.

### Fix

The fix depends on the root cause of the defect. For instance, suppose you refactor a function such that you use a local static variable in place of a global variable. In this case, the global variable is redundant and you can remove its declaration. Alternatively, if you are not sure if the global variable is used elsewhere, you can modify the name of the local static variable and all references within the function.

If the shadowing is intended and you do not want to fix the issue, add comments to your result or code to avoid another review. See

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Variable Shadowing Error

```
#include <stdio.h>

int fact[5]={1,2,6,24,120};

int factorial(int n)
 {
  int fact=1;
  /*Defect: Local variable hides global array with same name */

  for(int i=1;i<=n;i++)
    fact*=i;
  return(fact);
 }
```

Inside the `factorial` function, the integer variable `fact` hides the global integer array `fact`.

#### Correction — Change Variable Name

One possible correction is to change the name of one of the variables, preferably the one with more local scope.

```
#include <stdio.h>

int fact[5]={1,2,6,24,120};

int factorial(int n)
 {
  /* Fix: Change name of local variable */
  int f=1;

  for(int i=1;i<=n;i++)
    f*=i;

  return(f);
 }
```

### Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `VAR_SHADOWING`
**Impact:** Low

## Version History

**Introduced in R2013b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Write without a further read

Variable never read after assignment

## Description

This defect occurs when a value assigned to a variable is never read.

For instance, you write a value to a variable and then write a second value before reading the previous value. The first write operation is redundant.

### Risk

Redundant write operations often indicate programming errors. For instance, you forgot to read the variable between two successive write operations or unintentionally read a different variable.

### Fix

Identify the reason why you write to the variable but do not read it later. Look for common programming errors such as accidentally reading a different variable with a similar name.

If you determine that the write operation is redundant, remove the operation.

## Examples

### Write Without Further Read Error

```
void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();
    /* Defect: Useless write */
}
```

After the variable `level` gets assigned the value `4 * getsensor()`, it is not read.

#### Correction — Use Value After Assignment

One possible correction is to use the variable `level` after the assignment.

```
#include <stdio.h>

void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();

    /* Fix: Use level after assignment */
    printf("The value is %d", level);

}
```

The variable `level` is printed, reading the new value.

### Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `USELESS_WRITE`
**Impact:** Low

## Version History

**Introduced in R2013b**

### See Also

`Find defects (-checkers)` | `MISRA C:2012 Rule 2.2`

**Topics**

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Useless preprocessor conditional directive

Preprocessor conditional directive is always true or always false

## Description

This defect occurs when a preprocessor conditional directive always evaluates to the same logical value. For instance, you might be testing for the condition in a previous conditional directive. Consider this code:

```
long sensorFlag;
#if cond1
  sensorFlag = 0x0001;

#elif cond2
  sensorFlag = 0x00AA;

#else
    #if cond1 //Always false
        sensorFlag = 0xFF00;
    #endif
#endif
```

In the `#else` block, the conditional directive `#if cond1` is executed only when `cond1` is false. This directive is always `false`. Polyspace flags conditional directive that always evaluates to the same logical value.

### Risk

If a conditional directive is always true or always false, it might indicate an inadvertent error. For instance, the conditional directive might be unexpectedly impacted by a change elsewhere in your code. Alternatively, your conditional directive design might contain unexpected errors. In the preceding code, the statement `#if cond1` might be a typographical error. The developer intent might have been to use `#if cond3`.

### Fix

To fix this defect, remove the flagged conditional directive. If the useless conditional is unexpected, review and refactor your code.

## Examples

### Avoid Useless Preprocessor Conditional Directives

```
#define THE_ANSWER 42

#if defined(UNIVERSE)
    #define STARS 1
    #define MILKY_WAY 1
    #if defined(UNIVERSE) // Defect
        #define SUN 1
    #endif
```

```
#else
    #define STARS 0
    #define MILKY_WAY 0

    #if defined(UNIVERSE) // Defect
        #define SUN 0
    #endif
#endif
```

In this example, Polyspace flags two useless preprocessor conditional directives.

- The conditional directive `if defined(UNIVERSE)` on line six is placed inside the conditional block that starts with the same conditional. The flagged conditional is reached only when it is always `true`. Because this conditional is always `true`, you can remove it without changing code behavior. Polyspace flags the useless preprocessor directive.
- The conditional directive `if defined(UNIVERSE)` on line 13 is reached only when the same conditional evaluates to false. Because this conditional is always false, you can remove it without changing code behavior. Polyspace flags the useless preprocessor directive.

#### Correction — Remove Useless Preprocessor Conditional Directives

To fix this defect, remove the useless conditional directives. Alternatively, redesign your code. For instance, when the token `UNIVERSE` is undefined, you might set the value of `SUN` to zero without performing another check on `UNIVERSE`.

```
#define THE_ANSWER 42

#if defined(UNIVERSE)
    #define STARS 1
    #define MILKY_WAY 1
    #define SUN 1
#else
    #define STARS 0
    #define MILKY_WAY 0
    #define SUN 0
#endif
```

## Result Information

**Group:** Data flow
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `USELESS_PREPROC_CONDITION`
**Impact:** Low

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

**Topics**

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

13

# Security Defects

# Bad order of dropping privileges

Dropped higher elevated privileges before dropping lower elevated privileges

## Description

This defect occurs when you use functions such as `setuid` and `setgid` in the incorrect order, dropping higher elevated privileges before dropping lower elevated privileges. For example, you drop elevated primary group privileges before dropping elevated ancillary group privileges.

### Risk

If you drop privileges in the wrong order, you can potentially drop higher privileges that you need to drop lower privileges. The incorrect order can mean that privileges are not dropped compromising the security of your program.

### Fix

Respect this order of dropping elevated privileges:

- Drop (elevated) ancillary group privileges, then drop (elevated) primary group privileges.
- Drop (elevated) primary group privileges, then drop (elevated) user privileges.

## Examples

### Dropping User Privileges First

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()

static void sanitize_privilege_drop_check(uid_t olduid, gid_t oldgid)
{
    if (seteuid(olduid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
    if (setegid(oldgid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
}
void badprivilegedroporder(void) {
    uid_t
        newuid = getuid(),
        olduid = geteuid();
    gid_t
```

```
        newgid = getgid(),
        oldgid = getegid();

    if (setuid(newuid) == -1) {
        /* handle error condition */
        fatal_error();
    }
    if (setgid(newgid) == -1)  {
        /* handle error condition */
        fatal_error();
    }
    if (olduid == 0) {
        /* drop ancillary groups IDs only possible for root */
        if (setgroups(1, &newgid) == -1) {
            /* handle error condition */
            fatal_error();
        }
    }

    sanitize_privilege_drop_check(olduid, oldgid);
}
```

In this example, there are two privilege drops made in the incorrect order. `setgid` attempts to drop group privileges. However, `setgid` requires the user privileges, which were dropped previously using `setuid`, to perform this function. After dropping group privileges, this function attempts to drop ancillary groups privileges by using `setgroups`. This task requires the higher primary group privileges that were dropped with `setgid`. At the end of this function, it is possible to regain group privileges because the order of dropping privileges was incorrect.

#### Correction — Reverse Privilege Drop Order

One possible correction is to drop the lowest level privileges first. In this correction, ancillary group privileges are dropped, then primary group privileges are dropped, and finally user privileges are dropped.

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()

static void sanitize_privilege_drop_check(uid_t olduid, gid_t oldgid)
{
    if (seteuid(olduid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
    if (setegid(oldgid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
}
void badprivilegedroporder(void) {
    uid_t
        newuid = getuid(),
        olduid = geteuid();
    gid_t
        newgid = getgid(),
        oldgid = getegid();

    if (olduid == 0) {
        /* drop ancillary groups IDs only possible for root */
        if (setgroups(1, &newgid) == -1) {
            /* handle error condition */
            fatal_error();
        }
    }
    if (setgid(getgid()) == -1)  {
        /* handle error condition */
        fatal_error();
    }
    if (setuid(getuid()) == -1) {
        /* handle error condition */
        fatal_error();
    }

    sanitize_privilege_drop_check(olduid, oldgid);
}
```

### Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `BAD_PRIVILEGE_DROP_ORDER`
**Impact:** High

## Version History

**Introduced in R2016b**

### See Also

`Find defects (-checkers)`

**Topics**

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Deterministic random output from constant seed

Seeding routine uses a constant seed making the output deterministic

## Description

This defect occurs when you use standard random number generator functions that have deterministic output given a constant seed.

The checker detects this issue with the following random number generator functions:

- C Standard Library functions such as `srand`, `srandom` and `initstate`
- OpenSSL functions such as `RAND_seed` and `RAND_add`
- C++ Standard Library functions such as `std::linear_congruential_engine<>::seed()` and `std::mersenne_twister_engine<>::seed()` (and also the constructors of these class templates)

### Risk

With constant seeds, random number generator functions produce the same output every time your program is run. A hacker can disrupt your program if they know how your program behaves.

### Fix

Use a different random standard function or use a nonconstant seed.

Some standard random routines are inherently cryptographically weak on page 13-97, and should not be used for security purposes.

## Examples

### Random Number Generator Initialization

```
#include <stdlib.h>

void random_num(void)
{
    srand(12345U);
    /* ... */
}
```

This example initializes a random number generator using `srand` with a constant seed. The random number generation is deterministic, making this function cryptographically weak.

#### Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S
```

```
#include <stdlib.h>
#include <stdio.h>

unsigned int random_num_time(void)
{

    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `RAND_SEED_CONSTANT`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Predictable random output from predictable seed` | `Unsafe standard encryption function` | `Vulnerable pseudo-random number generator` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Errno not checked

errno is not checked for error conditions following function call

## Description

This defect occurs when you call a function that sets errno to indicate error conditions, but do not check errno after the call. For these functions, checking errno is the only reliable way to determine if an error occurred.

Functions that set errno on errors include:

- fgetwc, strtol, and wcstol.

  For a comprehensive list of functions, see documentation about errno.
- POSIX errno-setting functions such as encrypt and setkey.

### Risk

To see if the function call completed without errors, check errno for error values.

The return values of these errno-setting functions do not indicate errors. The return value can be one of the following:

- void
- Even if an error occurs, the return value can be the same as the value from a successful call. Such return values are called in-band error indicators.

You can determine if an error occurred only by checking errno.

For instance, strtol converts a string to a long integer and returns the integer. If the result of conversion overflows, the function returns LONG_MAX and sets errno to ERANGE. However, the function can also return LONG_MAX from a successful conversion. Only by checking errno can you distinguish between an error and a successful conversion.

### Fix

Before calling the function, set errno to zero.

After the function call, to see if an error occurred, compare errno to zero. Alternatively, compare errno to known error indicator values. For instance, strtol sets errno to ERANGE to indicate errors.

The error message in the Polyspace result shows the error indicator value that you can compare to.

## Examples

### errno Not Checked After Call to strtol

```
#include<stdio.h>
#include<stdlib.h>
```

```
#include<errno.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    long val = strtol(str, &endptr, base);
    printf("Return value of strtol() = %ld\n", val);
}
```

You are using the return value of `strtol` without checking `errno`.

**Correction — Check `errno` After Call**

Before calling `strtol`, set `errno` to zero . After a call to `strtol`, check the return value for `LONG_MIN` or `LONG_MAX` and `errno` for `ERANGE`.

```
#include<stdlib.h>
#include<stdio.h>
#include<errno.h>
#include<limits.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    errno = 0;
    long val = strtol(str, &endptr, base);
    if((val == LONG_MIN || val == LONG_MAX) && errno == ERANGE) {
         printf("strtol error");
         exit(EXIT_FAILURE);
    }
    printf("Return value of strtol() = %ld\n", val);
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `ERRNO_NOT_CHECKED`
**Impact:** Medium

# Version History

**Introduced in R2017a**

## See Also

`Find defects (-checkers)` | `Returned value of a sensitive function not checked` | `Errno not reset` | `Misuse of errno`

**Topics**

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Execution of a binary from a relative path can be controlled by an external actor

Command with relative path is vulnerable to malicious attack

## Description

This defect occurs when you call an external command with a relative path or without a path.

This defect also finds results that the **Execution of externally controlled command** defect checker finds.

### Risk

By using a relative path or no path to call an external command, your program uses an unsafe search process to find the command. An attacker can control the search process and replace the intended command with a command of their own.

### Fix

When you call an external command, specify the full path.

## Examples

### Call Command with Relative Path

```
# define _GNU_SOURCE
# include <sys/types.h>
# include <sys/socket.h>
# include <unistd.h>
# include <stdio.h>
# include <stdlib.h>
# include <wchar.h>
# include <string.h>
# define MAX_BUFFER 100

void rel_path()
{
    char * data;
    char data_buf[MAX_BUFFER] = "";
    data = data_buf;

    strcpy(data, "ls -la");
    FILE *pipe;
    pipe = popen(data, "wb");
    if (pipe != NULL) pclose(pipe);
}
```

In this example, Bug Finder flags `popen` because it tries to call `ls -la` using a relative path to the `ls` command. An attacker can manipulate the command to use a malicious version.

#### Correction — Use Full Path

One possible correction is to use the full path when calling the command.

```
# define _GNU_SOURCE
# include <sys/types.h>
# include <sys/socket.h>
# include <unistd.h>
# include <stdio.h>
# include <stdlib.h>
# include <wchar.h>
# include <string.h>
# define MAX_BUFFER 100

void rel_path()
{
    char * data;
    char data_buf[MAX_BUFFER] = "";
    data = data_buf;

    strcpy(data, "/usr/bin/ls -la");
    FILE *pipe;
    pipe = popen(data, "wb");
    if (pipe != NULL) pclose(pipe);
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `RELATIVE_PATH_CMD`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Load of library from a relative path can be controlled by an external actor` | `Vulnerable path manipulation` | `Execution of externally controlled command` | `Command executed from externally controlled path` | `Find defects (-checkers)`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# File access between time of check and use (TOCTOU)

File or folder might change state due to access race

## Description

This defect occurs when a race condition happens between checking the existence of a file or folder, and using the file or folder.

### Risk

An attacker can access and manipulate your file between your check for the file and your use of a file. Symbolic links are particularly risky because an attacker can change where your symbolic link points.

### Fix

Before using a file, do not check its status. Instead, use the file and check the results afterward.

## Examples

### Check File Before Using

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    if (access(log_path, W_OK)==0) {
        FILE* f = fopen(log_path, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

In this example, before opening and using the file, the function checks if the file exists. However, an attacker can change the file between the first and second lines of the function.

#### Correction — Open Then Check

One possible correction is to open the file, and then check the existence and contents afterward.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);
```

```
void toctou(char * log_path) {
    int fd = open(log_path, O_WRONLY);
    if (fd!=-1) {
        FILE *f = fdopen(fd, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TOCTOU`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Data race` | `Bad file access mode or status` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# File descriptor exposure to child process

Copied file descriptor used in multiple processes

## Description

This defect occurs when a process is forked and the child process uses file descriptors inherited from the parent process.

### Risk

When you fork a child process, file descriptors are copied from the parent process, which means that you can have concurrent operations on the same file. Use of the same file descriptor in the parent and child processes can lead to race conditions that may not be caught during standard debugging. If you do not properly manage the file descriptor permissions and privileges, the file content is vulnerable to attacks targeting the child process.

### Fix

Check that the file has not been modified before forking the process. Close all inherited file descriptors and reopen them with stricter permissions and privileges, such as read-only permission.

## Examples

### File Descriptor Accessed from Forked Process

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <fcntl.h>
# include <sys/types.h>
# include <sys/stat.h>



const char *test_file="/home/user/test.txt";

void func(void)
{
    char c;
    pid_t pid;
    /* create file descriptor in read and write mode */
    int fd = open(test_file, O_RDWR);
    if (fd == -1)
    {
        /* Handle error */
        abort();
    }
    /* fork process */
    pid = fork();
    if (pid == -1)
    {
        /* Handle error */
        abort();
    }
    else if (pid == 0)
    {   /* Child process accesses file descriptor inherited
        from parent process */
        (void)read(fd, &c, 1);
    }
    else
    {   /* Parent process access same file descriptor as
        child process */
        (void)read(fd, &c, 1);
    }
}
```

In this example, a file descriptor `fd` is created in read and write mode. The process is then forked. The child process inherits and accesses `fd` with the same permissions as the parent process. A race condition exists between the parent and child processes. The contents of the file is vulnerable to attacks through the child process.

**Correction — Close and Reopen Inherited File Descriptor**

After you create the file descriptor, check the file for tampering. Then, close the inherited file descriptor in the child process and reopen it in read-only mode.

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <fcntl.h>
# include <sys/types.h>
# include <sys/stat.h>


const char *test_file="/home/user/test.txt";

void func(void)
{
    char c;
    pid_t pid;

    /* Get the state of file for further file tampering checking */

    /* create file descriptor in read and write mode */
    int fd = open(test_file, O_RDWR);
    if (fd == -1)
    {
        /* Handle error */
        abort();
    }

    /* Be sure the file was not tampered with while opening */

    /* fork process */

    pid = fork();
    if (pid == -1)
    {
        /* Handle error */
        (void)close(fd);
        abort();
    }
    else if (pid == 0)
    {  /* Close file descriptor in child process and repoen
        it in read only mode */

        (void)close(fd);
        fd = open(test_file, O_RDONLY);
        if (fd == -1)
        {
            /* Handle error */
            abort();
        }


        (void)read(fd, &c, 1);
        (void)close(fd);
    }
    else
    {  /* Parent acceses original file descriptor */
        (void)read(fd, &c, 1);
        (void)close(fd);
    }
}
```

### Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `FILE_EXPOSURE_TO_CHILD`
**Impact:** Medium

## Version History

**Introduced in R2017b**

### See Also

`Find defects (-checkers)`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# File manipulation after chroot() without chdir("/")

Path-related vulnerabilities for file manipulated after call to chroot

## Description

This defect occurs when you have access to a file system outside of the jail created by chroot. By calling chroot, you create a file system jail that confines access to a specific file subsystem. However, this jail is ineffective if you do not call chdir("/").

### Risk

If you do not call chdir("/") after creating a chroot jail, file manipulation functions that takes a path as an argument can access files outside of the jail. An attacker can still manipulate files outside the subsystem that you specified, making the chroot jail ineffective.

### Fix

After calling chroot, call chdir("/") to make your chroot jail more secure.

## Examples

### Open File in chroot-jail

```
#include <unistd.h>
#include <stdio.h>

const char root_path[] = "/var/ftproot";
const char log_path[] = "file.log";
FILE* chrootmisuse() {
    FILE* res;
    chroot(root_path);
    chdir("base");
    res = fopen(log_path, "r");
    return res;
}
```

This example uses chroot to create a chroot-jail. However, to use the chroot jail securely, you must call chdir("\") afterward. This example calls chdir("base"), which is not equivalent. Bug Finder also flags fopen because fopen opens a file in the vulnerable chroot-jail.

#### Correction — Call chdir("/")

Before opening files, call chdir("/").

```
#include <unistd.h>
#include <stdio.h>

const char root_path[] = "/var/ftproot";
const char log_path[] = "file.log";
FILE* chrootmisuse() {
    FILE* res;
    chroot(root_path);
    chdir("/");
    res = fopen(log_path, "r");
    return res;
}
```

### Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** CHROOT_MISUSE
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

Umask used with chmod-style arguments | Vulnerable path manipulation | Find defects (-checkers)

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Function pointer assigned with absolute address

Constant expression is used as function address is vulnerable to code injection

## Description

This defect occurs when a function pointer is assigned an absolute address.

Bug Finder considers expressions with any combination of literal constants as an absolute address. The one exception is when the value of the expression is zero.

### Risk

Using a fixed address is not portable because it is possible that the address is invalid on other platforms.

An attacker can inject code at the absolute address, causing your program to execute arbitrary, possibly malicious, code.

### Fix

Do not use an absolute address with function pointers.

## Examples

### Function Pointer Address Assignment

```
extern int func0(int i, char c);
typedef int (*FuncPtr) (int, char);

FuncPtr funcptrabsoluteaddr() {
    return (FuncPtr)0x08040000;
}
```

In this example, the function returns a function pointer to the address `0x08040000`. If an attacker knows this absolute address, an attacker can compromise your program.

#### Correction — Function Address

One possible correction is to use the address of an existing function instead.

```
extern int func0(int i, char c);
typedef int (*FuncPtr) (int, char);

FuncPtr funcptrabsoluteaddr() {
    return &func0;
}
```

## Result Information

**Group:** Security
**Language:** C | C++

**Default:** Off
**Command-Line Syntax:** `FUNC_PTR_ABSOLUTE_ADDR`
**Impact:** Low

## Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Hard-coded sensitive data

Sensitive data is exposed in code, for instance as string literals

## Description

This defect occurs when data that is potentially sensitive is directly exposed in the code, for instance, as string literals. The checker identifies certain data as sensitive from their use in certain functions such as password encryption functions.

Following data can be potentially sensitive.

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Host name | • sethostname, setdomainname, gethostbyname, gethostbyname2, getaddrinfo, gethostbyname_r, gethostbyname2_r (string argument)<br>• inet_aton, inet_pton, inet_net_pton, inet_addr, inet_network (string argument)<br>• mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (2nd argument) |
| Password | • CreateProcessWithLogonW, LogonUser (1st argument)<br>• mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (3rd argument) |
| Database | • MySQL: mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (4th argument)<br>• SQLite: sqlite3_open, sqlite3_open16, sqlite3_open_v2 (1st argument)<br>• PostgreSQL: PQconnectdb<br>• Microsoft SQL: SQLDriverConnect (3rd argument) |
| User name | • getpw, getpwnam, getpwnam_r, getpwuid, getpwuid_r |
| Salt | crypt, crypt_r (2nd argument) |

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Cryptography keys and initialization vectors | OpenSSL:<br>- EVP_CipherInit, EVP_EncryptInit, EVP_DecryptInit (3rd argument)<br>- EVP_CipherInit_ex, EVP_EncryptInit_ex, EVP_DecryptInit_ex (4th argument) |
| Seed | - srand, srandom, initstate (1st argument)<br>- OpenSSL: RAND_seed, RAND_add |

#### Risk

Information that is hardcoded can be queried from binaries generated from the code.

#### Fix

Avoid hard coding sensitive information.

## Examples

### Sensitive Data Exposed Through String Literals

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"
const char *host = "localhost";
char *user = "guest";
char *passwd;

DbHandle connect_to_database_server(const char *db)
{
    passwd = (char*)"guest";
    return (DbHandle)
        mysql_real_connect (sql, host, user, passwd, db, 0, 0x0, 0);
}
```

In this example, the arguments host (host name), user (user name), and passwd (password) are string literals and directly exposed in the code.

Querying the generated binary for ASCII strings can reveal this information.

#### Correction – Read Sensitive Data from Secured Configuration Files

One possible correction is to read the data from a configuration file. In the following corrected example, the call to function `connect_to_database_server_init` presumably reads the host name, user name, and password into its arguments from a secured configuration file.

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"

int connect_to_database_server_init(const char **host,
                                    const char **user,
                                    const char **passwd,
                                    const char **db);

DbHandle connect_to_database_server(const char *db)
{
    const char *host_from_cfg;
    const char *user_from_cfg;
    const char *passwd_from_cfg;
    const char *db_from_cfg;
    if (connect_to_database_server_init(&host_from_cfg,
                                        &user_from_cfg,
                                        &passwd_from_cfg,
                                        &db_from_cfg))
    {
        return (DbHandle)
            mysql_real_connect (sql, host_from_cfg, user_from_cfg,
                        passwd_from_cfg, db_from_cfg,  0, 0x0, 0);
    }
    else
        return (DbHandle)0x0;
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `HARD_CODED_SENSITIVE_DATA`
**Impact:** Medium

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Inappropriate I/O operation on device files

Operation can result in security vulnerabilities or a system failure

## Description

This defect occurs when you do not check whether a file name parameter refers to a device file before you pass it to these functions:

- `fopen()`
- `fopen_s()`
- `freopen()`
- `remove()`
- `rename()`
- `CreateFile()`
- `CreateFileA()`
- `CreateFileW()`
- `_wfopen()`
- `_wfopen_s()`

Device files are files in a file system that provide an interface to device drivers. You can use these files to interact with devices.

**Inappropriate I/O operation on device files** does not raise a defect when:

- You use `stat` or `lstat`-family functions to check the file name parameter before calling the previously listed functions.
- You use a string comparison function to compare the file name against a list of device file names.

### Risk

Operations appropriate only for regular files but performed on device files can result in denial-of-service attacks, other security vulnerabilities, or system failures.

### Fix

Before you perform an I/O operation on a file:

- Use `stat()`, `lstat()`, or an equivalent function to check whether the file name parameter refers to a regular file.
- Use a string comparison function to compare the file name against a list of device file names.

## Examples

### Using `fopen()` Without Checking `file_name`

```
#include <stdio.h>
#include <string.h>
```

```
#define SIZE1024 1024

FILE* func()
{

    FILE* f;
    const char file_name[SIZE1024] = "./tmp/file";

    if ((f = fopen(file_name, "w")) == NULL) {
        /*handle error */
    };
    /*operate on file */
}
```

In this example, `func()` operates on the file `file_name` without checking whether it is a regular file. If `file_name` is a device file, attempts to access it can result in a system failure.

**Correction — Check File with `lstat()` Before Calling `fopen()`**

One possible correction is to use `lstat()` and the `S_ISREG` macro to check whether the file is a regular file. This solution contains a `TOCTOU` race condition that can allow an attacker to modify the file after you check it but before the call to fopen(). To prevent this vulnerability, ensure that `file_name` refers to a file in a secure folder.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>

#define SIZE1024 1024

FILE* func()
{

    FILE* f;
    const char file_name[SIZE1024] = "./tmp/file";
    struct stat orig_st;
    if ((lstat(file_name, &orig_st) != 0) ||
        (!S_ISREG(orig_st.st_mode))) {
        exit(0);
    }
    if ((f = fopen(file_name, "w")) == NULL) {
        /*handle error */
    };
    /*operate on file */
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `INAPPROPRIATE_IO_ON_DEVICE`
**Impact:** Medium

## Version History

**Introduced in R2018b**

## See Also

`File access between time of check and use (TOCTOU)` | `Opening previously opened resource` | `Resource leak` | `Returned value of a sensitive function not checked` | `Vulnerable path manipulation` | `Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Incorrect order of network connection operations

Socket is not correctly established due to bad order of connection steps or missing steps

## Description

This defect occurs when you perform operations on a network connection at the wrong point of the connection lifecycle.

### Risk

Sending or receiving data to an incorrectly connected socket can cause unexpected behavior or disclosure of sensitive information.

If you do not connect your socket correctly or change the connection by mistake, you can send sensitive data to an unexpected port. You can also get unexpected data from an incorrect socket.

### Fix

During socket connection and communication, check the return of each call and the length of the data.

Before reading, writing, sending, or receiving information, create sockets in this order:

- For a connection-oriented server socket (SOCK_STREAM or SOCK_SEQPACKET):

  ```
  socket(...);
  bind(...);
  listen(...);
  accept(...);
  ```
- For a connectionless server socket (SOCK_DGRAM):

  ```
  socket(...);
  bind(...);
  ```
- For a client socket (connection-oriented or connectionless):

  ```
  socket(...);
  connect(...);
  ```

## Examples

### Connecting a Connection-Oriented Server Socket

```
# include <stdio.h>
# include <string.h>
# include <time.h>
# include <arpa/inet.h>
# include <unistd.h>

enum { BUF_SIZE=1025 };

volatile int rd;
```

```
int stream_socket_server(int argc, char *argv[])
{
    int listenfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;

    char sendBuff[BUF_SIZE];
    time_t ticks;
    struct tm * timeinfo;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, 48, sizeof(serv_addr));
    memset(sendBuff, 48, sizeof(sendBuff));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(5000);

    bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

    listen(listenfd, 10);

    while(1)
    {
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);

        ticks = time(NULL);
        timeinfo = localtime(&ticks);
        strftime (sendBuff,BUF_SIZE,"%I:%M%p.",timeinfo);

        write(listenfd, sendBuff, strlen(sendBuff));

        close(connfd);
        sleep(1);
    }
}
```

This example creates a connection-oriented network connection. The function calls the correct functions in the correct order: `socket`, `bind`, `listen`, `accept`. However, the program should write to the `connfd` socket instead of the `listenfd` socket.

**Correction — Use Safe Socket**

One possible correction is to write to the `connfd` function instead of the `listenfd` socket.

```
# include <stdio.h>
# include <string.h>
# include <time.h>
# include <arpa/inet.h>
# include <unistd.h>

enum { BUF_SIZE=1025 };

volatile int rd;

int stream_socket_server_good(int argc, char *argv[])
{
    int listenfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;

    char sendBuff[BUF_SIZE];
    time_t ticks;
    struct tm * timeinfo;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, 48, sizeof(serv_addr));
    memset(sendBuff, 48, sizeof(sendBuff));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(5000);

    bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    listen(listenfd, 10);

    while(1)
    {
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
        ticks = time(NULL);
        timeinfo = localtime(&ticks);
        strftime (sendBuff,BUF_SIZE,"%I:%M%p.",timeinfo);
        write(connfd, sendBuff, strlen(sendBuff));
        close(connfd);
        sleep(1);
    }
}
```

### Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `BAD_NETWORK_CONNECT_ORDER`
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Information leak via structure padding

Padding bytes can contain sensitive information

## Description

This defect occurs when you do not initialize the padding data of a structure or union before passing it across a trust boundary. A compiler adds padding bytes to the structure or union to ensure a proper memory alignment of its members. The bit-fields of the storage units can also have padding bits.

**Information leak via structure padding** raises a defect when:

- You call an untrusted function with structure or union pointer type argument containing uninitialized padding data.

  All external functions are considered untrusted.
- You copy or assign a structure or union containing uninitialized padding data to an untrusted object.

  All external structure or union objects, the output parameters of all externally linked functions, and the return pointer of all external functions are considered untrusted objects.

### Risk

The padding bytes of the passed structure or union might contain sensitive information that an untrusted source can access.

### Fix

- Prevent the addition of padding bytes for memory alignment by using the `pack` pragma or attribute supported by your compiler.
- Explicitly declare and initialize padding bytes as fields within the structure or union.
- Explicitly declare and initialize bit-fields corresponding to padding bits, even if you use the `pack` pragma or attribute supported by your compiler.

## Examples

### Structure with Padding Bytes Passed to External Function

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>

typedef struct s_padding
{
 /* Padding bytes may be introduced between
 * 'char c' and 'int i'
 */
        char c;
    int i;
/*Padding bits may be introduced around the bit-fields
* even if you use "#pragma pack" (Windows) or
* __attribute__((__packed__)) (GNU)*/

    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;



/* External function */
extern void copy_object(void *out, void *in, size_t s);

void func(void *out_buffer)
{
/*Padding bytes not initialized*/

    S_Padding s = {'A', 10, 1, 3, {}};
/*Structure passed to external function*/

    copy_object((void *)out_buffer, (void *)&s, sizeof(s));
}

void main(void)
{
    S_Padding s1;
    func(&s1);
}
```

In this example, structure `s1` can have padding bytes between the `char c` and `int i` members. The bit-fields of the storage units of the structure can also contain padding bits. The content of the padding bytes and bits is accessible to an untrusted source when `s1` is passed to `func`.

**Correction — Use pack Pragma to Prevent Padding Bytes**

One possible correction in Microsoft Visual Studio is to use `#pragma pack()` to prevent padding bytes between the structure members. To prevent padding bits in the bit-fields of `s1`, explicitly declare and initialize the bit-fields even if you use `#pragma pack()`.

```
 #include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define CHAR_BIT 8

#pragma pack(push, 1)

typedef struct s_padding
{
/*No Padding bytes when you use "#pragma pack" (Windows) or
* __attribute__((__packed__)) (GNU)*/
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
/* Padding bits explicitely declared */
```

```
    unsigned int bf_filler : sizeof(unsigned) * CHAR_BIT - 3;
    unsigned char buffer[20];
}

    S_Padding;

#pragma pack(pop)


/* External function */
extern void copy_object(void *out, void *in, size_t s);



void func(void *out_buffer)
{
    S_Padding s = {'A', 10, 1, 3, 0 /* padding bits */, {}};
    copy_object((void *)out_buffer, (void *)&s, sizeof(s));
}

void main(void)
{
    S_Padding s1;
    func(&s1);
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `PADDING_INFO_LEAK`
**Impact:** Low

# Version History

**Introduced in R2018a**

## See Also

`Find defects (-checkers)` | `Memory comparison of padding data` | `Use of memset with size argument zero` | `Invalid assumptions about memory organization` | `Sensitive heap memory not cleared before release` | `Uncleared sensitive data in stack`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# LDAP injection

Data read from an untrusted source is used in the construction of an LDAP query

## Description

This defect occurs when data read from an untrusted source such as standard input is used in the construction of an LDAP query.

This defect checker detects the flow of data from an untrusted source to a function that constructs an LDAP query. The checker recognizes OpenLDAP search functions such as `ldap_search()`, `ldap_search_ext()`, and so on. In all search functions, the checker considers the second parameter (base) and the fourth parameter (search filter) as sensitive to untrusted inputs. Untrusted sources can include read operations on the standard input `stdin` using the `fread()` or `fgets()` function.

Note that the defect checker is not available in the Polyspace user interface and is disabled even if you select the value `all` for the option `Find defects (-checkers)`. For the issue to be detected, the checker must be enabled explicitly using the option `-checkers LDAP_INJECTION`.

### Risk

If you construct the search base or search filters in an LDAP query from user inputs but use the inputs directly in the LDAP query without any validation or sanitization, they are vulnerable to LDAP injection. A malicious user can inject unintended LDAP queries masquerading as input, which can compromise secure user information.

For instance, in this LDAP search filter, the variable *`username`* is obtained from user inputs. The search filter is vulnerable to LDAP injection.

`(&(name=`*`username`*`)(pass=`*`password`*`))`

If the user enters `*)(name=*))(|(name=*` for *`username`*, the search filter becomes:

`(&(name=*)(name=*))(|(name=*)(pass=`*`password`*`))`

The resulting search filter is trivially true and bypasses the authentication mechanism built into the search filter using the variable *`password`*.

### Fix

Validate user inputs if you use them in the construction of the search base or search filters in an LDAP query.

If you validate user inputs using dedicated functions, you can make the LDAP injection checker aware of your validation functions. Suppose that the function `ldap_validate_inputs` checks user inputs for malicious entries.

```
int ldap_validate_inputs(
     char *,    /* String to check */
     int);      /* Length of string */
```

Suppose that the *n*-th parameter of this function is the user input to validate. For instance, the first argument in the above signature could be the string to validate.

To make the LDAP injection checker aware of this function:

1. In a file with extension `.dl`, add:

   ```
   .include "models/interfaces/ldap.dl"

   Ldap.Basic.sanitizing("ldap_validate_inputs", $OutParameterDeref(n-1)).
   ```

   If *n* is 1 (that is, the first parameters of the function is the parameter of interest), then the statement becomes:

   ```
   .include "models/interfaces/ldap.dl"

   Ldap.Basic.sanitizing("ldap_validate_inputs", $OutParameterDeref(0)).
   ```
2. Specify this file using the option `-code-behavior-specifications`. For instance, if the file is named `ldapAdditionalFunctions.dl`, use the analysis option:

   ```
   -code-behavior-specifications ldapAdditionalFunctions.dl
   ```

## Examples

### LDAP Filter Created from Untrusted User Input

```
#include <stdio.h>
#include <stddef.h>
#include <sys/time.h>
#include <ldap.h>

int LDAPSearch()
{
    const char* hostName = "fabrikam.com";
    LDAP* ld;

    // Creating LDAP search parameters
    ld = ldap_open(hostName, LDAP_PORT);

    const char* base = "DC=fabrikam,DC=com";
    const int scope = LDAP_SCOPE_SUBTREE;
    char* attrs[6];
    attrs[0] = "cn";
    attrs[1] = "company";
    attrs[2] = "department";
    attrs[3] = "telephoneNumber";
    attrs[4] = "memberOf";
    attrs[5] = NULL;
    const int attrsonly = 0;

    // Creating LDAP search filter from untrusted source
    char filter[1024] = "";
    char idToSearch[512];
    char password[512];

    fread(idToSearch, sizeof(idToSearch), 1, stdin);
    fread(password, sizeof(password), 1, stdin);
```

```
    strcat(filter, "(&(uid=");
    strcat(filter, idToSearch);
    strcat(filter, ")(pass=");
    strcat(filter, password);
    strcat(filter, "))");

    // Performing LDAP search
    ldap_search(ld, base, scope, filter, attrs, attrsonly);

    return 0;
}
```

In this example, the fourth argument to `ldap_search()`, that is, the variable `filter` (representing the LDAP search filter), is constructed from untrusted user inputs. The user inputs comes from the variables `idToSearch` and `password`.

**Correction – Validate User Input Before Use in LDAP Query**

You can write your own sanitization function to validate or sanitize the input before using the input to construct an LDAP search filter. In this example, assume that the function `ldap_validate_inputs()` validates the user input and removes elements that do not belong to an allowlist.

```
#include <stdio.h>
#include <stddef.h>
#include <sys/time.h>
#include <ldap.h>

extern void ldap_validate_inputs(char* s, size_t n);

int LDAPSearch()
{
    const char* hostName = "fabrikam.com";
    LDAP* ld;

    // Creating LDAP search parameters
    ld = ldap_open(hostName, LDAP_PORT);

    const char* base = "DC=fabrikam,DC=com";
    const int scope = LDAP_SCOPE_SUBTREE;
    char* attrs[6];
    attrs[0] = "cn";
    attrs[1] = "company";
    attrs[2] = "department";
    attrs[3] = "telephoneNumber";
    attrs[4] = "memberOf";
    attrs[5] = NULL;
    const int attrsonly = 0;

    // Creating LDAP search filter from untrusted source
    char filter[1024] = "";
    char idToSearch[512];
    char password[512];

    fread(idToSearch, sizeof(idToSearch), 1, stdin);
    fread(password, sizeof(password), 1, stdin);
    strcat(filter, "(&(uid=");
    strcat(filter, idToSearch);
    strcat(filter, ")(pass=");
    strcat(filter, password);
    strcat(filter, "))");

    // Sanitize filter
    ldap_validate_inputs(filter, sizeof(filter));

    // Performing LDAP search
    ldap_search(ld, base, scope, filter, attrs, attrsonly);

    return 0;
}
```

To make the Polyspace analysis aware of the nature of your sanitization function:

1. Specify this code in a file with extension `.dl` (for instance, `sanitizeFunctions.dl`):

   ```
   .include "models/interfaces/ldap.dl"

   Ldap.Basic.sanitizing("ldap_validate_inputs", $OutParameterDeref(0)).
   ```
2. Specify this option with the analysis:

   ```
   -code-behavior-specifications sanitizeFunctions.dl
   ```

   See also `-code-behavior-specifications`.

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `LDAP_INJECTION`
**Impact:** High

# Version History

**Introduced in R2023a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Load of library from a relative path can be controlled by an external actor

Library loaded with relative path is vulnerable to malicious attacks

## Description

This defect occurs when library loading routines that load an external library use a relative path or do not use a path at all.

### Risk

By using a relative path or no path to load an external library, your program uses an unsafe search process to find the library. An attacker can control the search process and replace the intended library with a library of their own.

### Fix

When you load an external library, specify the full path.

## Examples

### Open Library with Library Name

```
#include <dlfcn.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>

void relative_path()
{
    dlopen("liberty.dll",RTLD_LAZY);
}
```

In this example, `dlopen` opens the `liberty` library by calling only the name of the library. However, this call to the library uses a relative path to find the library, which is unsafe.

#### Correction — Use Full Path to Library

One possible correction is to use the full path to the library when you load it into your program.

```
#include <dlfcn.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>

void relative_path()
{
    dlopen("/home/my_libs/library/liberty.dll",RTLD_LAZY);
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `RELATIVE_PATH_LIB`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Execution of a binary from a relative path can be controlled by an external actor` | `Vulnerable path manipulation` | `Library loaded from externally controlled path` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Mismatch between data length and size

Data size argument is not computed from actual data length

## Description

This defect occurs when you do not check the length argument and data buffer argument of memory copying functions such as `memcpy`, `memset`, or `memmove`, to protect against buffer overflows.

### Risk

If an attacker can manipulate the data buffer or length argument, the attacker can cause buffer overflow by making the actual data size smaller than the length.

This mismatch in length allows the attacker to copy memory past the data buffer to a new location. If the extra memory contains sensitive information, the attacker can now access that data.

This defect is similar to the SSL Heartbleed bug.

### Fix

When copying or manipulating memory, compute the length argument directly from the data so that the sizes match.

## Examples

### Copy Buffer of Data

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    memcpy(&(beta.data[num]), os->data + 2, length);

    return(1);
}
```

This function copies the buffer `alpha` into a buffer `beta`. However, the `length` variable is not related to `data+2`.

#### Correction — Check Buffer Length

One possible correction is to check the length of your buffer against the maximum value minus 2. This check ensures that you have enough space to copy the data to the `beta` structure.

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    if (length<(os->max -2)) {
        memcpy(&(beta.data[num]), os->data + 2, length);
    }

    return(1);

}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `DATA_LENGTH_MISMATCH`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Copy of overlapping memory` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing case for switch condition

switch variable not covered by cases and default case is missing

## Description

This defect occurs when the switch variable can take values that are not covered by a case statement.

**Note** Bug Finder only raises a defect if the switch variable is not full range.

### Risk

If the switch variable takes a value that is not covered by a case statement, your program can have unintended behavior.

A switch-statement that makes a security decision is particularly vulnerable when all possible values are not explicitly handled. An attacker can use this situation to deviate the normal execution flow.

### Fix

It is good practice to use a default statement as a catch-all for values that are not covered by a case statement. Even if the switch variable takes an unintended value, the resulting behavior can be anticipated.

## Examples

### Missing Default Condition

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}
```

```
int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    }

    printf("Welcome!\n");
    return r;
}
```

In this example, the `enum` parameter `User` can take a value `UNKNOWN` that is not covered by a `case` statement.

#### Correction — Add a Default Condition

One possible correction is to add a default condition for possible values that are not covered by a `case` statement.

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}

int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    break;
    default:
        printf("Invalid login credentials!\n");
    }

    printf("Welcome!\n");
    return r;
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `MISSING_SWITCH_CASE`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Misuse of readlink()

Third argument of `readlink` does not leave space for null terminator in buffer

## Description

This defect occurs when you pass a buffer size argument to `readlink()` that does not leave space for a null terminator in the buffer.

For instance:

```
ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf));
```

The third argument is exactly equal to the size of the second argument. For large enough symbolic links, this use of `readlink()` does not leave space to enter a null terminator.

### Risk

The `readlink()` function copies the content of a symbolic link (first argument) to a buffer (second argument). However, the function does not append a null terminator to the copied content. After using `readlink()`, you must explicitly add a null terminator to the buffer.

If you fill the entire buffer when using `readlink`, you do not leave space for this null terminator.

### Fix

When using the `readlink()` function, make sure that the third argument is one less than the buffer size.

Then, append a null terminator to the buffer. To determine where to add the null terminator, check the return value of `readlink()`. If the return value is -1, an error has occurred. Otherwise, the return value is the number of characters (bytes) copied.

## Examples

### Incorrect Size Argument of `readlink`

```
#include <unistd.h>

#define SIZE1024 1024

extern void display_path(const char *);

void func() {
    char buf[SIZE1024];
    ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf));
    if (len > 0) {
        buf[len - 1] = '\0';
    }
    display_path(buf);
}
```

In this example, the third argument of `readlink` is exactly the size of the buffer (second argument). If the first argument is long enough, this use of `readlink` does not leave space for the null terminator.

Also, if no characters are copied, the return value of `readlink` is 0. The following statement leads to a buffer underflow when `len` is 0.

```
buf[len - 1] = '\0';
```

**Correction — Make Sure Size Argument is One Less Than Buffer Size**

One possible correction is to make sure that the third argument of `readlink` is one less than size of the second argument.

The following corrected code also accounts for `readlink` returning 0.

```
#include <stdlib.h>
#include <unistd.h>

#define fatal_error() abort()
#define SIZE1024 1024

extern void display_path(const char *);

void func() {
    char buf[SIZE1024];
    ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf) - 1);
    if (len != -1) {
        buf[len] = '\0';
        display_path(buf);
    }
    else {
        /* Handle error */
        fatal_error();
    }
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `READLINK_MISUSE`
**Impact:** Medium

# Version History

**Introduced in R2017a**

## See Also

Find defects (-checkers) | Returned value of a sensitive function not checked | Invalid use of standard library string routine | Array access out of bounds | Pointer access out of bounds | File access between time of check and use (TOCTOU)

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Predictable random output from predictable seed

Seeding routine uses a predictable seed making the output predictable

## Description

This defect occurs when you use standard random number generator functions with a nonconstant but predictable seed. Examples of predictable seed generators are `time`, `gettimeofday`, and `getpid`.

The checker detects this issue with the following random number generator functions:

- C Standard Library functions such as `srand`, `srandom` and `initstate`
- C++ Standard Library functions such as `std::linear_congruential_engine<>::seed()` and `std::mersenne_twister_engine<>::seed()` (and also the constructors of these class templates)

### Risk

When you use predictable seed values for random number generation, your random numbers are also predictable. A hacker can disrupt your program if they know how your program behaves.

### Fix

You can use a different function to generate less predictable seeds.

You can also use a different random number generator that does not require a seed. For example, the Windows API function `rand_s` seeds itself by default. It uses information from the entire system, for example, system time, thread ids, system counter, and memory clusters. This information is more random and a user cannot access this information.

Some standard random routines are inherently cryptographically weak on page 13-97, and should not be used for security purposes.

## Examples

### Seed as an Argument

```
#include <stdlib.h>
#include <time.h>

void seed_rng(int seed)
{
    srand(seed);
}

int generate_num(void)
{
    seed_rng(time(NULL) + 3);
    /* ... */
}
```

This example uses `srand` to start the random number generator with `seed` as the seed. However, `seed` is predictable because the function `time` generates it. So, an attacker can predict the random numbers generated by `srand`.

#### Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int generate_num(void)
{
    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

### Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `RAND_SEED_PREDICTABLE`
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Deterministic random output from constant seed` | `Unsafe standard encryption function` | `Vulnerable pseudo-random number generator` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Privilege drop not verified

Attacker can gain unintended elevated access to program

## Description

This defect occurs when you relinquish privileges using functions such as `setuid` but do not verify that the privileges were actually dropped before exiting your function.

### Risk

If privilege relinquishment fails, an attacker can regain elevated privileges and have more access to your program than intended. This security hole can cause unexpected behavior in your code if left open.

### Fix

Before the end of scope, verify that the privileges that you dropped were actually dropped.

## Examples

### Drop Privileges Within a Function

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()
extern int need_more_privileges;

void missingprivilegedropcheck()
{
    /* Code intended to run with elevated privileges */

    /* Temporarily drop elevated privileges */
    if (seteuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges */

    if (need_more_privileges) {
        /* Restore elevated privileges */
        if (seteuid(0) != 0) {
            /* Handle error */
            fatal_error();
        }
        /* Code intended to run with elevated privileges */
    }

    /* ... */
    /* Permanently drop elevated privileges */
    if (setuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges */
}
```

In this example, privileges are elevated and dropped to run code with the intended privilege level. When privileges are dropped, the privilege level before exiting the function body is not verified. A malicious attacker can regain their elevated privileges.

**Correction — Verify Privilege Drop**

One possible correction is to use `setuid` to verify that the privileges were dropped.

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()
extern int need_more_privileges;

void missingprivilegedropcheck()
{
    /* Store the privileged ID for later verification */
    uid_t privid = geteuid();

    /* Code intended to run with elevated privileges   */

    /* Temporarily drop elevated privileges */
    if (seteuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges  */

    if (need_more_privileges) {
        /* Restore elevated Privileges */
        if (seteuid(privid) != 0) {
            /* Handle error */
            fatal_error();
        }
        /* Code intended to run with elevated privileges   */
    }

    /* ... */

    /* Restore privileges if needed */
    if (geteuid() != privid) {
        if (seteuid(privid) != 0) {
            /* Handle error */
            fatal_error();
        }
    }

    /* Permanently drop privileges */
    if (setuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    if (setuid(0) != -1) {
        /* Privileges can be restored, handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges; */
}
```

### Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `MISSING_PRIVILEGE_DROP_CHECK`
**Impact:** High

## Version History

**Introduced in R2016b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Returned value of a sensitive function not checked

Sensitive functions called without checking for unexpected return values and errors

## Description

This defect occurs when you call sensitive standard functions that return information about possible errors and you do one of the following:

- Ignore the return value.

  You simply do not assign the return value to a variable, or explicitly cast the return value to `void`.
- Use an output from the function (return value or argument passed by reference) without testing the return value for errors.

The checker considers a function as sensitive if the function call is prone to failure because of reasons such as:

- Exhausted system resources (for example, when allocating resources).
- Changed privileges or permissions.
- Tainted sources when reading, writing, or converting data from external sources.
- Unsupported features despite an existing API.

The checker only considers functions where the *return value* indicates if the function completed without errors.

Some of these functions can perform critical tasks such as:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

### Risk

If you do not check the return value of functions that perform sensitive tasks and indicate error information through their return values, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

### Fix

Before continuing with the program, test the return value of critical sensitive functions.

For sensitive functions that are not critical, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

## Examples

### Sensitive Function Return Ignored

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);//Noncompliant
}
int read_file(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r");
  return 0; //Noncompliant

}
```

This example shows calls to the sensitive POSIX functions `pthread_attr_init` and `fmemopen`. Their return values are ignored, causing defect.

#### Correction — Cast Function to (void)

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);//Compliant
}
int read_file(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  (void)fmemopen (argv[1], strlen (argv[1]), "r"); //Compliant

  return 0;
}
```

#### Correction — Test Return Value

One possible correction is to test the return value of `pthread_attr_init` and `fmemopen` to check for errors.

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

void initialize() {
    pthread_attr_t attr;

    int result = pthread_attr_init(&attr);//Compliant
    if(result != 0){
        //Handle fatal error
    }
}
int read_file(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r");
  if (in==NULL){
      // Handle error
  }
  return 0;//Compliant
}
```

### Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0));
    pthread_join(thread_id,  &res);
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

#### Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

### Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `RETURN_NOT_CHECKED`
**Impact:** High

## Version History

**Introduced in R2016b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Sensitive data printed out

Function prints sensitive data

## Description

This defect occurs when print functions such as `stdout` or `stderr` print sensitive information.

The checker considers the following as sensitive information:

- Return values of password manipulation functions such as `getpw`, `getpwnam` or `getpwuid`.
- Input values of functions such as the Windows-specific function `LogonUser`.

### Risk

Printing sensitive information, such as passwords or user information, allows an attacker additional access to the information.

### Fix

One fix for this defect is to not print out sensitive information.

If you are saving your logfile to an external file, set the file permissions so that attackers cannot access the logfile information.

## Examples

### Printing Passwords

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

extern void verify_null(const char* buf);
void bug_sensitivedataprint(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    puts("Name\n");
    puts(pwd.pw_name);
    puts("PassWord\n");
    puts(pwd.pw_passwd);
    memset(buf, 0, sizeof(buf));
    verify_null(buf);
}
```

In this example, Bug Finder flags `puts` for printing out the password `pwd.pw_passwd`.

#### Correction — Obfuscate the Password

One possible correction is to obfuscate the password information so that the information is not visible.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

extern void verify_null(const char* buf);

void sensitivedataprint(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    puts("Name\n");
    puts(pwd.pw_name);
    puts("PassWord\n");
    puts("XXXXXXXX\n");
    memset(buf, 0, sizeof(buf));
    verify_null(buf);
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SENSITIVE_DATA_PRINT`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Sensitive heap memory not cleared before release` | `Uncleared sensitive data in stack` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Sensitive heap memory not cleared before release

Sensitive data not cleared or released by memory routine

## Description

This defect occurs when dynamically allocated memory contains sensitive data and you do not clear the data before you free the memory.

### Risk

If the memory zone is reallocated, an attacker can still inspect the sensitive data in the old memory zone.

### Fix

Before calling `free`, clear out the sensitive data using `memset` or `SecureZeroMemory`.

## Examples

### Sensitive Buffer Freed, Not Cleared

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    free(buf);
}
```

In this example, the function uses a buffer of passwords and frees the memory before the end of the function. However, the data in the memory is not cleared by using the `free` command.

#### Correction — Nullify Data

One possible correction is to write over the data to clear out the sensitive information. This example uses `memset` to write over the data with zeros.

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0;i<(sizeof(arr)/sizeof(arr[0]));i++) assert(arr[i]==0)
```

```
void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);

    if (buf) {
        getpwnam_r(my_user, &pwd, buf, bufsize, &result);
        memset(buf, 0, (size_t)1024);
        isNull(buf);
        free(buf);
    }
}
```

### Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SENSITIVE_HEAP_NOT_CLEARED`
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Uncleared sensitive data in stack` | `Sensitive data printed out` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# SQL injection

Data read from an untrusted source is used in the construction of an SQL query

## Description

This defect occurs when data read from an untrusted source such as standard input is used in the construction of an SQL query.

This defect checker detects the flow of data from an untrusted source to a function that executes an SQL query (or prepares an SQL query to execute later). The checker flags functions from the following API-s:

- MySQL C API – SQL query functions such as `mysql_query()` and `mysql_real_query()`.
- SQLite API – SQL query functions such as `sqlite3_exec()` or SQL statement preparation functions such as `sqlite3_prepare()` and `sqlite3_prepare_v2()`.

Untrusted sources include strings read from the standard input `stdin` using the `fread()` or `fgets()` function or from files opened with `fopen()`.

Note that the defect checker is not available in the Polyspace user interface and is disabled even if you select the value `all` for the option `Find defects (-checkers)`. For the issue to be detected, the checker must be enabled explicitly using the option `-checkers SQL_INJECTION`.

### Risk

If you construct SQL queries from user inputs but use the inputs directly in the query construction without any sanitization, the queries are vulnerable to SQL injection. A malicious user can inject code masquerading as input, resulting in:

- Bypassing of logic that secures part of the database.
- Execution of malicious SQL queries or shell commands.

For instance, this SQL query, where the variables *`username`* and *`itemName`* are obtained from user inputs, is vulnerable to SQL injection:

```
SELECT * FROM items WHERE owner = 'userName' AND itemname = 'itemName';
```

If a malicious user enters two inputs such as `john` and `anItem' OR 'a' = 'a`, the query translates to:

```
SELECT * FROM items WHERE owner = 'john' AND itemname = 'anItem' OR 'a' = 'a';
```

The query effectively means that the `WHERE` clause is always true. This allows the user to access the entire database even if they do not have the requisite permissions.

### Fix

Depending on the API you are using, follow these sanitization approaches to sanitize user inputs before using them in SQL queries. For instance:

- MySQL C API – Escape values in the SQL query string using `mysql_real_escape_string()` or `mysql_real_escape_string_quote()`.
- SQLite API – Create parametrized queries using the functions `sqlite3_prepare_*()` with parameters and then bind the parameters using the functions `sqlite3_bind_*()`.

You can also write your own sanitization functions to validate or clean up the user input. For information on how to make the checker aware of custom sanitization functions, see the next section.

### Extend Checker

You can extend this checker by specifying your own SQL statement execution and sanitization functions.

Suppose you want to specify that:

- Function `sql_exec` executes an SQL statement or prepares an SQL statement for execution:

  ```
  int sql_exec
      (char*,           /* SQL query */
       sqlite3* ,       /* Database */
       int (*callback)(void*,int,char**,char**),
       void*,
       char**);
  ```

  Suppose the *`n_exec`*-th argument of this function is the SQL query string. For instance, the first argument in the above signature could be the SQL query string.
- Function `sql_sanitize` sanitizes the untrusted (tainted) data that is later used in an SQL statement.

  ```
  int sql_sanitize(
       char *,    /* String to sanitize */
       int);      /* Length of string */
  ```

  Suppose the *`n_sanitize`*-th argument of this function is the data string to sanitize. For instance, the first argument in the above signature could be the data to sanitize.

To make the SQL injection checker aware of these functions:

1. In a file with extension `.dl`, add this code:

   ```
   .include "models/interfaces/sql.dl"

   Sql.Basic.execution("sql_exec", $InParameterDeref(n_exec-1)).
   Sql.Basic.sanitizing("sql_sanitize", $OutParameterDeref(n_sanitize-1)).
   ```

   If *`n_exec`* and *`n_sanitize`* are both 1 (that is, the first parameters of each function are the parameters of interest), then the statements become:

   ```
   .include "models/interfaces/sql.dl"

   Sql.Basic.execution("sql_exec", $InParameterDeref(0)).
   Sql.Basic.sanitizing("sql_sanitize", $OutParameterDeref(0)).
   ```
2. Specify this file using the option `-code-behavior-specifications`. For instance, if the file is named `sqlAdditionalFunctions.dl`, use the analysis option:

   ```
   -code-behavior-specifications sqlAdditionalFunctions.dl
   ```

## Examples

### Execution of SQL Query Created from Untrusted User Input

```
#include <stdio.h>
#include <string.h>
#include <sqlite3.h>

// Function to insert names into database
void insertIntoDatabase(void)
{
    sqlite3* db;
    char query[256] = "";
    char name[256];
    char* error_message;
    int nameLength;

    sqlite3_open("userCredentials.db", &db);

    fread(name, 1, 128, stdin);

    strcat(query, "INSERT INTO privatenames (name) VALUES ('");
    strcat(query, name);
    strcat(query, "');");


    if (sqlite3_exec(db, query, NULL, NULL, &error_message))
        {
            sqlite3_free(error_message);
        }
}
```

In this example, the function `sqlite3_exec()` creates an SQL query by concatenating parts of an SQL command with a user input. The final query is the following:

```
INSERT INTO privatenames (name) VALUES ('name')
```

A malicious user can pass SQL commands inside the user input so that the overall SQL query performs additional operations (other SQL operations or shell commands). For instance, if the user passes an input like this:

```
john'); DELETE FROM privatenames;--
```

The SQL query becomes:

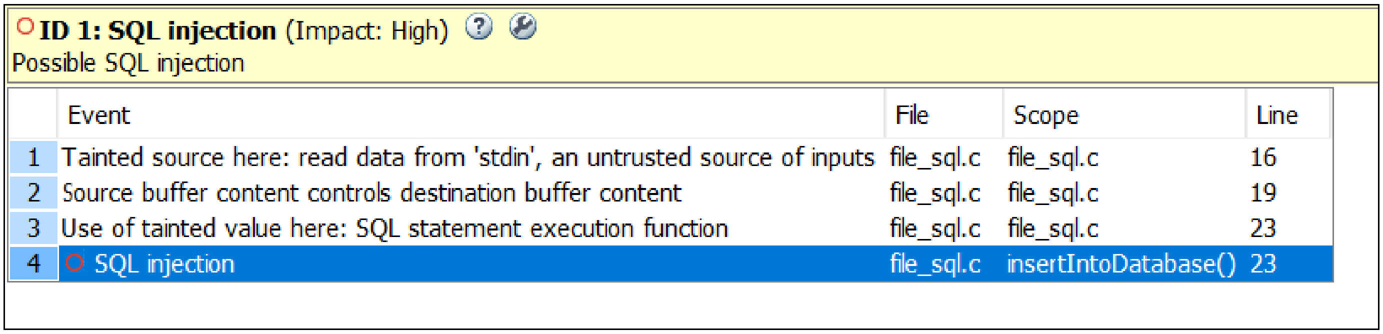
```
INSERT INTO privatenames (name) VALUES ('john');
DELETE FROM privatenames;
--');
```

The `--` results in the remainder of the line being commented out. The resulting SQL query consists of two statements: the first statement is the query that you intended and the second one results in an unintended deletion of all items from the database.

The event list on the **Result Details** pane shows how tainted data propagates into an SQL query. For instance, in this example:

- The first event points to where the tainted data is obtained.
- The second event points to where the tainted data flows to another variable.
- The third event points to where the tainted data is eventually used.

Click on an event to navigate to the corresponding location in the source code.



○ **ID 1: SQL injection** (Impact: High)

Possible SQL injection

| | Event | File | Scope | Line |
|---|---|---|---|---|
| 1 | Tainted source here: read data from 'stdin', an untrusted source of inputs | file_sql.c | file_sql.c | 16 |
| 2 | Source buffer content controls destination buffer content | file_sql.c | file_sql.c | 19 |
| 3 | Use of tainted value here: SQL statement execution function | file_sql.c | file_sql.c | 23 |
| 4 | ○ SQL injection | file_sql.c | insertIntoDatabase() | 23 |

**Correction – Write Custom Sanitization Function**

You can write your own sanitization function to validate or sanitize the input before using the input to construct an SQL query. In this example, the function `sanitizeString()` performs some basic sanitization such as removing semicolons from the input string.

```
#include <stdio.h>
#include <string.h>
#include <sqlite3.h>

void sanitizeString(char *, int);

// Function to insert names into database
void insertIntoDatabase(void)
{
    sqlite3* db;
    char query[256] = "";
    char name[256];
    char* error_message;
    int nameLength;

    sqlite3_open("userCredentials.db", &db);

    fread(name, 1, 128, stdin);

    nameLength = sizeof(name)/sizeof(char);
    sanitizeString(name, nameLength);

    strcat(query, "INSERT INTO privatenames (name) VALUES (");
    strcat(query, name);
    strcat(query, ");");


    if (sqlite3_exec(db, query, NULL, NULL, &error_message))
        {
            sqlite3_free(error_message);
        }
}
```

```
void sanitizeString(char *name, int len)
{
    for (int i = 0; i < len; i++)
        {
            // Remove semicolons
            if(name[i] == ';')
                {
                    name[i] = ' ';
                }
            // Other sanitization of string
        }
}
```

To make the Polyspace analysis aware of the nature of your sanitization function:

1 Specify this code in a file with extension `.dl` (for instance, `sanitizeFunctions.dl`):

   ```
   .include "models/interfaces/sql.dl"

   Sql.Basic.sanitizing("sanitizeString", $OutParameterDeref(0)).
   ```

2 Specify this option with the analysis:

   ```
   -code-behavior-specifications sanitizeFunctions.dl
   ```

   See also `-code-behavior-specifications`.

### Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SQL_INJECTION`
**Impact:** High

## Version History

**Introduced in R2023a**

### See Also

`Find defects (-checkers)`

**Topics**

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Umask used with chmod-style arguments

Argument to umask allows external user too much control

## Description

This defect occurs when umask commands have arguments specified in the style of arguments to chmod and provide possibly unintended permissions. For instance:

- The umask command provides more permissions to the group than the current user.
- The umask command provides more permissions to other users than the group.

For new files, the umask argument or the mask value specifies which permissions *not* to set, in other words, which permissions to remove. The mask is bitwise-negated and then applied to new file permissions. In contrast, chmod sets the permissions as you specify them.

### Risk

If you use chmod-style arguments, you specify opposite permissions of what you want. This mistake can give external users unintended read/write access to new files and folders.

### Fix

To fix this defect, perform both of these tasks:

- Enable read permission for the user.
- Set the mask value so that the user (u) has equal or fewer permissions turned off than the group (g) and the group has equal or fewer permissions turned off than other users (o), or u <= g <= o.

You can see the umask value by calling,

```
umask
```

or the symbolic value by calling,

```
umask -S
```

## Examples

### Setting the Default Mask

```
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>

typedef mode_t (*umask_func)(mode_t);

const mode_t default_mode = (
    S_IRUSR    /* 00400 */
    | S_IWUSR  /* 00200 */
    | S_IRGRP  /* 00040 */
    | S_IWGRP  /* 00020 */
    | S_IROTH  /* 00004 */
    | S_IWOTH  /* 00002 */
    );         /* 00666 (i.e. -rw-rw-rw-) */

static void my_umask(mode_t mode)
{
    umask(mode);
}

int umask_use(mode_t m)
{
    my_umask(default_mode);
    return 0;
}
```

This example uses a function called `my_umask` to set the default mask mode. However, the `default_mode` variable gives the permissions 666 or `-rw-rw-rw`. `umask` negates this value. However, this negation means the default mask mode turns off read/write permissions for the user, group users, and other outside users.

#### Correction — Negate Preferred Permissions

One possible correction is to negate the `default_mode` argument to `my_umask`. This correction nullifies the negation `umask` for new files.

```
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>

typedef mode_t (*umask_func)(mode_t);

const mode_t default_mode = (
    S_IRUSR    /* 00400 */
    | S_IWUSR  /* 00200 */
    | S_IRGRP  /* 00040 */
    | S_IWGRP  /* 00020 */
    | S_IROTH  /* 00004 */
    | S_IWOTH  /* 00002 */
    );         /* 00666 (i.e. -rw-rw-rw-) */

static void my_umask(mode_t mode)
{
    umask(mode);
}

int umask_use(mode_t m)
{
    my_umask(~default_mode);
    return 0;
}
```

## Result Information

**Group:** Security

**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `BAD_UMASK`
**Impact:** Low

## Version History

**Introduced in R2015b**

## See Also

`Vulnerable permission assignments` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

### External Websites

`umask` — Linux Manual Page

# Uncleared sensitive data in stack

Variable in stack is not cleared and contains sensitive data

## Description

This defect occurs when statically allocated memory contains sensitive data and you do not clear the data before exiting a function or program.

### Risk

Leaving sensitive information in your stack, such as passwords or user information, allows an attacker additional access to the information after your program has ended.

### Fix

Before exiting a function or program, clear out the memory zones that contain sensitive data by using `memset` or `SecureZeroMemory`.

## Examples

### Static Buffer of Password Information

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>

void bug_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
}
```

In this example, a static buffer is filled with password information. The program frees the stack memory at the end of the program. However, the data is still accessible from the memory.

#### Correction — Clear Memory

One possible correction is to write over the memory before exiting the function. This example uses `memset` to clear the data from the buffer memory.

```
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0;i<(sizeof(arr)/sizeof(arr[0]));i++) assert(arr[i]==0)

void corrected_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    memset(buf, 0, (size_t)1024);
    isNull(buf);
}
```

### Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SENSITIVE_STACK_NOT_CLEARED`
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Sensitive heap memory not cleared before release` | `Sensitive data printed out` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unsafe call to a system function

Unsanitized command argument has exploitable vulnerabilities

## Description

This defect occurs when you use a function that invokes an implementation-defined command processor. These functions include:

- The C standard `system()` function.
- The POSIX `popen()` function.
- The Windows `_popen()` and `_wpopen()` functions.

### Risk

If the argument of a function that invokes a command processor is not sanitized, it can cause exploitable vulnerabilities. An attacker can execute arbitrary commands or read and modify data anywhere on the system.

### Fix

Do not use a `system`-family function to invoke a command processor. Instead, use safer functions such as POSIX `execve()` and WinAPI `CreateProcess()`.

## Examples

### `system()` Called

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};

void func(char *arg)
{
    char buf[SIZE512];
    int retval=sprintf(buf, "/usr/bin/any_cmd %s", arg);

    if (retval<=0 || retval>SIZE512){
        /* Handle error */
        abort();
    }
    /* Use of system() to pass any_cmd with
    unsanitized argument to command processor */

    if (system(buf) == -1) {
    /* Handle error */
```

```
  }
}
```

In this example, `system()` passes its argument to the host environment for the command processor to execute. This code is vulnerable to an attack by command-injection.

#### Correction — Sanitize Argument and Use execve()

In the following code, the argument of `any_cmd` is sanitized, and then passed to `execve()` for execution. `exec`-family functions are not vulnerable to command-injection attacks.

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};


void func(char *arg)
{
  char *const args[SIZE3] = {"any_cmd", arg, NULL};
  char  *const env[] = {NULL};

  /* Sanitize argument */

  /* Use execve() to execute any_cmd. */

  if (execve("/usr/bin/time", args, env) == -1) {
    /* Handle error */
  }
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `UNSAFE_SYSTEM_CALL`
**Impact:** High

# Version History

**Introduced in R2017b**

## See Also

`Command executed from externally controlled path` | `Execution of externally controlled command` | `Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”

“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Unsafe standard encryption function

Function is not reentrant or uses a risky encryption algorithm

## Description

This defect occurs when a standard encryption function uses a broken or weak cryptographic algorithm. For example, `crypt` is not reentrant and is based on the risky Data Encryption Standard (DES).

### Risk

The use of a broken, weak, or nonstandard algorithm can expose sensitive information to an attacker. A determined hacker can access the protected data using various techniques.

If the weak function is nonreentrant, when you use the function in concurrent programs, there is an additional race condition risk.

### Fix

Avoid functions that use these encryption algorithms. Instead, use a reentrant function that uses a stronger encryption algorithm.

**Note** Some implementations of `crypt` support additional, possibly more secure, encryption algorithms.

## Examples

### Decrypting Password Using `crypt`

```
#define _GNU_SOURCE
#include <pwd.h>
#include <string.h>
#include <crypt.h>

volatile int rd = 1;

const char *salt = NULL;
struct crypt_data input, output;

int verif_pwd(const char *pwd, const char *cipher_pwd, int safe)
{
    int r = 0;
    char *decrypted_pwd = NULL;

    switch(safe)
    {
      case 1:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;
      case 2:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      default:
        decrypted_pwd = crypt(pwd, cipher_pwd);
        break;
    }

    r = (strcmp(cipher_pwd, decrypted_pwd) == 0);

    return r;
}
```

In this example, `crypt_r` and `crypt` decrypt a password. However, `crypt` is nonreentrant and uses the unsafe Data Encryption Standard algorithm.

**Correction — Use `crypt_r`**

One possible correction is to replace `crypt` with `crypt_r`.

```
#define _GNU_SOURCE
#include <pwd.h>
#include <string.h>
#include <crypt.h>

volatile int rd = 1;

const char *salt = NULL;
struct crypt_data input, output;

int verif_pwd(const char *pwd, const char *cipher_pwd, int safe)
{
    int r = 0;
    char *decrypted_pwd = NULL;

    switch(safe)
    {
      case 1:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      case 2:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      default:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;
    }

    r = (strcmp(cipher_pwd, decrypted_pwd) == 0);

    return r;
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `UNSAFE_STD_CRYPT`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Deterministic random output from constant seed` | `Predictable random output from predictable seed` | `Vulnerable pseudo-random number generator` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unsafe standard function

Function unsafe for security-related purposes

## Description

This defect occurs when you use standard functions that are unsafe and must not be used for security-related programming. Functions can be unsafe for many reasons. Some functions are unsafe because they are nonreentrant. Other functions change behavior depending on the target or platform, making some implementations unsafe.

### Risk

Some unsafe functions are not reentrant, meaning that the contents of the function are not locked during a call. So, an attacker can change the values midstream.

`getlogin` specifically can be unsafe depending on the implementation. Some implementations of `getlogin` return only the first eight characters of a log-in name. An attacker can use a different login with the same first eight characters to gain entry and manipulate the program.

### Fix

Avoid unsafe functions for security-related purposes. If you cannot avoid unsafe functions, use a safer version of the function instead. For `getlogin`, use `getlogin_r`.

## Examples

### Using `getlogin`

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <string.h>
#include <stdlib.h>


volatile int rd = 1;

int login_name_check(char *user)
{
    int r = -2;
    char *name = getlogin();
    if (name != NULL)
    {
        if (strcmp(name, user) == 0)
        {
            r = 0;
        }
        else
            r = -1;
    }
    return r;
}
```

This example uses getlogin to compare the user name of the current user to the given user name . However, getlogin can return something other than the current user name because a parallel process can change the string.

#### Correction — Use getlogin_r

One possible correction is to use getlogin_r instead of getlogin. getlogin_r is reentrant, so you can trust the result.

```
#define _POSIX_C_SOURCE 199506L // use of getlogin_r
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <string.h>
#include <stdlib.h>


volatile int rd = 1;

enum {  NAME_MAX_SIZE=64  };

int login_name_check(char *user)
{
    int r;
    char name[NAME_MAX_SIZE];

    if (getlogin_r(name, sizeof(name)) == 0)
    {
        if ((strlen(user) < sizeof(name)) &&
                     (strncmp(name, user, strlen(user)) == 0))
        {
            r = 0;
        }
        else
            r = -1;
    }
    else
        r = -2;
    return r;
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** UNSAFE_STD_FUNC
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Use of obsolete standard function` | `Use of dangerous standard function` | `Invalid use of standard library string routine` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Use of dangerous standard function

Dangerous functions cause possible buffer overflow in destination buffer

## Description

This issue occurs when your code uses standard functions that write data to a buffer in a way that can result in buffer overflows.

The following table lists dangerous standard functions, the risks of using each function, and what function to use instead. The checker flags:

- Any use of an inherently dangerous function.
- An use of a possibly dangerous function only if the size of the buffer to which data is written can be determined at compile time. The checker does not flag an use of such a function with a dynamically allocated buffer.

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `gets` | Inherently dangerous — You cannot control the length of input from the console. | `fgets` |
| `std::cin::operator>>` and `std::wcin::operator>>` | Inherently dangerous — You cannot control the length of input from the console. | Preface calls to `cin` by `cin.width` to control the input length. This method can result in truncated input.<br>To avoid potential buffer overflow and truncated input, use `std::string` objects as destinations for >> operator. |
| `strcpy` | Possibly dangerous — If the size of the destination buffer is too small to accommodate the source buffer and a null terminator, a buffer overflow might occur. | Use the function `strlen()` to determine the size of the source buffer, and allocate sufficient memory so that the destination buffer can accommodate the source buffer and a null terminator. Instead of `strcpy`, use the function `strncpy`. |
| `stpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `stpncpy` |
| `lstrcpy` or `StrCpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy` |
| `strcat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `strncat`, `strlcat`, or `strcat_s` |

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| lstrcat or StrCat | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | StringCbCat, StringCchCat, strncay, strcat_s, or strlcat |
| wcpcpy | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | wcpncpy |
| wcscat | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | wcsncat, wcslcat, or wcncat_s |
| wcscpy | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | wcsncpy |
| sprintf | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | snprintf |
| vsprintf | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | vsnprintf |

#### Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

#### Fix

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Examples

### Using sprintf

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
```

```
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

**Correction — Use `snprintf` with Buffer Size**

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

### Destination Buffer Too Small

In this example, the size of the `source` buffer is unknown, while the size of the `destination` buffer is fixed at `128`. The size of the `destination` buffer might not be sufficient to accommodate the characters from the `source` buffer and terminate the buffer with a null. Polyspace reports a violation of the rule.

```
#include <string.h>

int main(int argc, char *argv[]) {
  const char *const source = (argc && argv[0]) ? argv[0] : "";
  char destination[128];
  strcpy(source, destination);

  return 0;
}
```

### Correction — Allocate Sufficient Memory for Destination

This violation is resolved by allocating sufficient memory for the `destination` buffer. For instance, use the function `strlen()` to calculate the size of the `source` buffer and allocate sufficient memory for the `destination` buffer so that it can accommodate all characters from the `source` buffer and the null terminator (`'\0'` ).

```
#include <string.h>

int main(int argc, char *argv[]) {
  const char *const source = (argc && argv[0]) ? argv[0] : "";
  char* destination = (char *)malloc(strlen(source)+ 1);
  if(destination!=NULL){
      strncpy(source, destination,sizeof(source));//No defect
  }else{
      /*Handle Error*/
  }
  //...
  free(destination);
  return 0;
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `DANGEROUS_STD_FUNC`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Use of obsolete standard function` | `Unsafe standard function` | `Invalid use of standard library string routine` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Use of non-secure temporary file

Temporary generated file name not secure

## Description

This defect occurs when you use temporary file routines that are not secure.

### Risk

If an attacker guesses the file name generated by a standard temporary file routine, the attacker can:

- Cause a race condition when you generate the file name.
- Precreate a file of the same name, filled with malicious content. If your program reads the file, the attacker's file can inject the malicious code.
- Create a symbolic link to a file storing sensitive data. When your program writes to the temporary file, the sensitive data is deleted.

### Fix

To create temporary files, use a more secure standard temporary file routine, such as `mkstemp` from POSIX.1-2001.

Also, when creating temporary files with routines that allow flags, such as `mkostemp`, use the exclusion flag `O_EXCL` to avoid race conditions.

## Examples

### Temp File Created With `tempnam`

```
#define _BSD_SOURCE
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int test_temp()
{
    char tpl[] = "abcXXXXXX";
    char suff_tpl[] = "abcXXXXXXsuff";
    char *filename = NULL;
    int fd;

    filename = tempnam("/var/tmp", "foo_");

    if (filename != NULL)
    {
        printf("generated tmp name (%s) in (%s:%s:%s)\n",
               filename, getenv("TMPDIR") ? getenv("TMPDIR") : "$TMPDIR",
               "/var/tmp", P_tmpdir);

        fd = open(filename, O_CREAT, S_IRWXU|S_IRUSR);
        if (fd != -1)
        {
            close(fd);
            unlink(filename);
            return 1;
        }
    }
    return 0;
}
```

In this example, Bug Finder flags `open` because it tries to use an unsecure temporary file. The file is opened without exclusive privileges. An attacker can access the file causing various risks on page 13-84.

#### Correction — Add O_EXCL Flag

One possible correction is to add the `O_EXCL` flag when you open the temporary file.

```
#define _BSD_SOURCE
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int test_temp()
{
    char tpl[] = "abcXXXXXX";
    char suff_tpl[] = "abcXXXXXXsuff";
    char *filename = NULL;
    int fd;

    filename = tempnam("/var/tmp", "foo_");

    if (filename != NULL)
    {
        printf("generated tmp name (%s) in (%s:%s:%s)\n",
               filename, getenv("TMPDIR") ? getenv("TMPDIR") : "$TMPDIR",
               "/var/tmp", P_tmpdir);

        fd = open(filename, O_CREAT|O_EXCL, S_IRWXU|S_IRUSR);
        if (fd != -1)
        {
            close(fd);
            unlink(filename);
            return 1;
        }
    }
    return 0;
}
```

## Result Information

**Group:** Security  
**Language:** C | C++  
**Default:** Off  
**Command-Line Syntax:** `NON_SECURE_TEMP_FILE`  
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Data race` | `Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”  
“Interpret Bug Finder Results in Polyspace Access Web Interface”  
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”  
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Use of obsolete standard function

Obsolete routines can cause security vulnerabilities and portability issues

## Description

This defect occurs when you use standard function routines that are considered legacy, removed, deprecated, or obsolete by C/C++ coding standards.

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| `asctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |
| `asctime_r` | Deprecated in POSIX.1-2008 | Implementation based on unsafe function `sprintf`. | `strftime` or `asctime_s` |
| `bcmp` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcmp` |
| `bcopy` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcpy` or `memmove` |
| `brk` and `sbrk` | Marked as legacy in SUSv2 and POSIX.1-2001. | | `malloc` |
| `bsd_signal` | Removed in POSIX.1-2008 | | `sigaction` |
| `bzero` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | `memset` |
| `ctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |
| `ctime_r` | Deprecated in POSIX.1-2008 | Implementation based on unsafe function `sprintf`. | `strftime` or `asctime_s` |
| `cuserid` | Removed in POSIX.1-2001. | Not reentrant. Precise functionality not standardized causing portability issues. | `getpwuid` |
| `ecvt` and `fcvt` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | Not reentrant | `snprintf` |
| `ecvt_r` and `fcvt_r` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | | `snprintf` |
| `ftime` | Removed in POSIX.1-2008 | | `time`, `gettimeofday`, `clock_gettime` |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| gamma, gammaf, gammal | Function not specified in any standard because of historical variations | Portability issues. | tgamma, lgamma |
| gcvt | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | snprintf |
| getcontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| getdtablesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_OPEN _MAX ) |
| gethostbyaddr | Removed in POSIX.1-2008 | Not reentrant | getaddrinfo |
| gethostbyname | Removed in POSIX.1-2008 | Not reentrant | getnameinfo |
| getpagesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_PAGE SIZE ) |
| getpass | Removed in POSIX.1-2001. | Not reentrant. | getpwuid |
| getw | Not present in POSIX.1-2001. | | fread |
| getwd | Marked legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | getcwd |
| index | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strchr |
| makecontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| memalign | Appears in SunOS 4.1.3. Not in 4.4 BSD or POSIX.1-2001 | | posix_memalign |
| mktemp | Removed in POSIX.1-2008. | Generated names are predictable and can cause a race condition. | mkstemp removes race risk |
| pthread_attr_ getstackaddr and pthread_attr_ setstackaddr | | Ambiguities in the specification of the stackaddr attribute cause portability issues | pthread_attr_ getstack and pthread_attr_ setstack |
| putw | Not present in POSIX.1-2001. | Portability issues. | fwrite |
| qecvt and qfcvt | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | snprintf |
| qecvt_r and qfcvt_r | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | snprintf |
| rand_r | Marked as obsolete in POSIX.1-2008 | | |
| re_comp | BSD API function | Portability issues | regcomp |
| re_exes | BSD API function | Portability issues | regexec |
| rindex | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strrchr |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| scalb | Removed in POSIX.1-2008 | | scalbln, scalblnf, or scalblnl |
| sigblock | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigmask | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigsetmask | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigstack | Interface is obsolete and not implemented on most platforms. | Portability issues. | sigaltstack |
| sigvec | 4.3BSD signal API whose origin is unclear | | sigaction |
| swapcontext | Removed in POSIX.1-2008 | Portability issues. | Use POSIX threads. |
| tmpnam and tmpnam_r | Marked as obsolete in POSIX.1-2008. | This function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined. | mkstemp, tmpfile |
| ttyslot | Removed in POSIX.1-2001. | | |
| ualarm | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | Errors are under-specified | setitimer or POSIX timer_create |
| usleep | Removed in POSIX.1-2008. | | nanosleep |
| utime | SVr4, POSIX.1-2001. POSIX.1-2008 marks as obsolete. | | |
| valloc | Marked as obsolete in 4.3BSD.<br>Marked as legacy in SUSv2.<br>Removed from POSIX.1-2001 | | posix_memalign |
| vfork | Removed from POSIX.1-2008 | Under-specified in previous standards. | fork |
| wcswcs | This function was not included in the final ISO/IEC 9899:1990/ Amendment 1:1995 (E). | | wcsstr |
| WinExec | WinAPI provides this function only for 16-bit Windows compatibility. | | CreateProcess |
| LoadModule | WinAPI provides this function only for 16-bit Windows compatibility. | | CreateProcess |

#### Fix

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Printing Out Time

```
#include <stdio.h>
#include <time.h>

void timecheck_bad(int argc, char *argv[])
{
    time_t ticks;

    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks));
}
```

In this example, the function `ctime` formats the current time and prints it out. However, `ctime` was removed after C99 because it does not work on multithreaded programs.

#### Correction — Different Time Function

One possible correction is to use `strftime` instead because this function uses a set buffer size.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

void timecheck_good(int argc, char *argv[])
{
    char outBuff[1025];
    time_t ticks;
    struct tm * timeinfo;

    memset(outBuff, 0, sizeof(outBuff));

    ticks = time(NULL);
    timeinfo = localtime(&ticks);
    strftime(outBuff,sizeof(outBuff),"%I:%M%p.",timeinfo);
    fprintf(stdout, outBuff);
}
```

## Result Information

**Group:** Security

**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `OBSOLETE_STD_FUNC`
**Impact:** Low
**Tags**: #deprecatedFunctions

## Version History

**Introduced in R2015b**

### See Also

`Use of dangerous standard function` | `Unsafe standard function` | `Invalid use of standard library string routine` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Vulnerable path manipulation

Path argument with `/../`, `/abs/path/`, or other unsecure elements

## Description

This defect occurs when you create a relative or absolute path from a tainted source and you then use the path to open/create files.

### Risk

Relative path elements, such as `".."` can resolve to locations outside the intended folder. Absolute path elements, such as `"/abs/path"` can also resolve to locations outside the intended folder.

An attacker can use these types of path traversal elements to traverse to the rest of the file system and access other files or folders.

### Fix

Avoid vulnerable path traversal elements such as `/../` and `/abs/path/`. Use fixed file names and locations wherever possible.

## Examples

### Relative Path Traversal

```
# include <stdio.h>
# include <string.h>
# include <wchar.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <unistd.h>
# include <stdlib.h>
# define BASEPATH "/tmp/"
# define FILENAME_MAX 512

static void Relative_Path_Traversal(void)
{
    char * data;
    char data_buf[FILENAME_MAX] = BASEPATH;
    char sub_buf[FILENAME_MAX];

    if (fgets(sub_buf, FILENAME_MAX, stdin) == NULL) exit (1);
    data = data_buf;
    strcat(data, sub_buf);

    FILE *file = NULL;
    file = fopen(data, "wb+");
    if (file != NULL) fclose(file);
}
```

```
int path_call(void){
    Relative_Path_Traversal();
}
```

This example opens a file from `"/tmp/"`, but uses a relative path to the file. An external user can manipulate this relative path when `fopen` opens the file.

**Correction — Use Fixed File Name**

One possible correction is to use a fixed file name instead of a relative path. This example uses `file.txt`.

```
# include <stdio.h>
# include <string.h>
# include <wchar.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <unistd.h>
# include <stdlib.h>
# define BASEPATH "/tmp/"
# define FILENAME_MAX 512

static void Relative_Path_Traversal(void)
{
    char * data;
    char data_buf[FILENAME_MAX] = BASEPATH;
    data = data_buf;

    /* FIX: Use a fixed file name */
    strcat(data, "file.txt");
    FILE *file = NULL;
    file = fopen(data, "wb+");
    if (file != NULL) fclose(file);
}

int path_call(void){
    Relative_Path_Traversal();
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `PATH_TRAVERSAL`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Use of path manipulation function without maximum-sized buffer checking` | `Find defects (-checkers)`

**Topics**

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Vulnerable permission assignments

Argument gives read/write/search permissions to external users

## Description

This defect occurs when functions that can change resource permissions, such as `chmod`, `umask`, `creat`, or `open`, specify permissions that allow unintended actors to modify or read the resource.

### Risk

If you give outside users or outside groups a wider range or permissions than required, you potentially expose your sensitive information and your modifications. This defect is especially dangerous for permissions related to:

- Program configurations
- Program executions
- Sensitive user data

### Fix

Set your permissions so that the user (`u`) has more permissions than the group (`g`), and so the group has more permissions than other users (`o`), or `u >= g >= o`.

## Examples

### Create File with Other Permissions

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void bug_dangerouspermissions(const char * log_path) {
    mode_t mode = S_IROTH | S_IXOTH | S_IWOTH;
    int fd = creat(log_path, mode);

    if (fd) {
        write(fd, "Hello\n", 6);
    }
    close(fd);
    unlink(log_path);
}
```

In this example, the `log_path` file is created with more rights for the other outside users, than the current user. The permissions are `---------rwx`.

#### Correction — Modify User Permissions

One possible correction is to modify the user permissions for the file. In this correction, the user has read/write/execute permissions, but other users do not.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void corrected_dangerouspermissions(const char * log_path) {
    mode_t mode = S_IRUSR | S_IXUSR | S_IWUSR;
    int fd = creat(log_path, mode);

    if (fd) {
        write(fd, "Hello\n", 6);
    }
    close(fd);
    unlink(log_path);
}
```

### Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `DANGEROUS_PERMISSIONS`
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Umask used with chmod-style arguments` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Vulnerable pseudo-random number generator

Using a cryptographically weak pseudo-random number generator

## Description

This defect occurs when you use cryptographically weak pseudo-random number generator (PRNG) routines.

The list of cryptographically weak routines flagged by this checker include:

- `rand`, `random`
- `drand48`, `lrand48`, `mrand48`, `erand48`, `nrand48`, `jrand48`, and their `_r` equivalents such as `drand48_r`
- `RAND_pseudo_bytes`

### Risk

These cryptographically weak routines are predictable and must not be used for security purposes. When a predictable random value controls the execution flow, your program is vulnerable to malicious attacks.

### Fix

Use more cryptographically sound random number generators, such as `CryptGenRandom` (Windows), `OpenSSL/RAND_bytes`(Linux/UNIX).

## Examples

### Random Loop Numbers

```
#include <stdio.h>
#include <stdlib.h>

volatile int rd = 1;
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;

    nloops = rand();

    for (j = 0; j < nloops; j++) {
        if (random_r(&buf, &i))
            exit(1);
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

This example uses rand and random_r to generate random numbers. If you use these functions for security purposes, these PRNGs can be the source of malicious attacks.

#### Correction — Use Stronger PRNG

One possible correction is to replace the vulnerable PRNG with a stronger random number generator.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rand.h>

volatile int rd = 1;
int main(int argc, char* argv[])
{
    int j, r, nloops;
    unsigned char buf;
    unsigned int seed;
    int i = 0;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <seed> <nloops>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    seed = atoi(argv[1]);
    nloops = atoi(argv[2]);

    for (j = 0; j < nloops; j++) {
        if (RAND_bytes(&buf, i) != 1)
            exit(1);
        printf("RAND_bytes: %u\n", (unsigned)buf);
    }
    return 0;
}
```

### Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** VULNERABLE_PRNG
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

Deterministic random output from constant seed | Predictable random output from predictable seed | Unsafe standard encryption function | Find defects (-checkers)

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"

"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Critical data member is not private

A critical data member is declared `public`

## Description

This defect occurs when you declare a critical nonstatic data member of a class to be `public`. By default, Polyspace assumes that no data member is critical. Specify the critical data members in your code by using the code behavior `CRITICAL_DATA`. See “Specifying Critical Data Members”. If you do not specify any critical data members, Polyspace raises a warning during the analysis.

### Risk

Declaring the critical data members as `public` allows the clients of a class to modify critical data members. You can inadvertently introduce vulnerabilities when critical data members are public. The vulnerabilities of such code are difficult to find and time-consuming to fix.

### Fix

To fix this defect, determine which data members are critical and declare them as `private` or `protected`.

## Examples

### Declare Critical Data Members as Private

```
#include <string.h>
#define MAX_PASSWORD_LENGTH 15
#define MAX_USERNAME_LENGTH 15

class UserAccount
{
public:
  UserAccount(char *username, char *password)
  {
    //...
  }

  int authorizeAccess(char *username, char *password)
  {
    //...
  }

  char username[MAX_USERNAME_LENGTH+1];
  char password[MAX_PASSWORD_LENGTH+1];
};
```

In this example, the data members `username` and `password` are declared as `public`. Specify these variables as critical in a code behavior XML file:

```
<specifications>
   <members>
```

```
    <member name="password" kind="variable">
        <behavior name="CRITICAL_DATA"/>
    </member>
    <member name="username" kind="variable">
        <behavior name="CRITICAL_DATA"/>
    </member>
   </members>
</specifications>
```

After you specify the variables as critical, Polyspace flags the public critical data members. If you do not specify the critical data members, Polyspace assumes that no data members are critical and the defect is not raised.

### Correction — Declare Critical Variables as Private

To fix this defect, declare the critical variables as `private`

```
#include <string.h>


#define MAX_PASSWORD_LENGTH 15
#define MAX_USERNAME_LENGTH 15

class UserAccount
{
public:
  UserAccount(char *username, char *password)
  {
    //...
  }

  int authorizeAccess(char *username, char *password)
  {
    //...
  }
private:
  char username[MAX_USERNAME_LENGTH+1];
  char password[MAX_PASSWORD_LENGTH+1];
};
```

## Result Information

**Group:** Security
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `CRITICAL_DATA_MEMBER_DECLARED_PUBLIC`
**Impact:** High

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

**Topics**

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Uncertain memory cleaning

The code clears information that might be sensitive from memory but compiler optimization might leave the information untouched

## Description

This defect occurs when you clean memory by writing `0` into it but the memory is not read after the cleaning operation. Because the memory is not read after the cleaning, a compiler might skip the cleaning operation when optimizing the code. For instance, consider this code:

```
void foo(){
    void* buffer = getSensitiveData();
    //...
    memset(buffer, 0, sizeof(buffer));
}
```

Here, `buffer` might contain sensitive data. The call to `memset()` is intended to scrub the data. Because buffer is not read after the scrubbing, the compiler optimization process might ignore the `memset()` command. The data in `buffer` might remain uncleaned. Polyspace flags such uncertain memory cleaning. Polyspace does not raise this defect if either of these conditions are true:

- The return value of the memory cleaning function is assigned to a variable.
- The memory cleaning function scrubs a global object.

### Risk

When a variable is written with no further read, compilers might consider it as a deadstore. To optimize the code, the compiler might remove any assignments to deadstore variables.

If such code optimization removes code that scrubs sensitive information from memory, then sensitive information might remain uncleaned in your system. An attacker can then access the sensitive information and use it to further erode the protection mechanisms in your code.

### Fix

When scrubbing memory, use code that the compiler cannot remove. For instance, instead of `memset`, use secure functions, such as `memset_s` or `fill_n`. If you use an older version of C or C++ where these functions are unavailable, consider updating your code to a more recent version. Alternatively, you might implement your own secure memory-scrubbing function. When using your own implementation, review the compiled code to make sure that the memory-cleaning operations are untouched by the compiler.

## Examples

### Avoid Uncertainty in Cleaning Sensitive Memory

```
#include <string.h>

extern int getSensitiveData(char*, size_t);
extern int writeDB(char*, char*);
char Gbuffer[64];
void Job(char *key) {
    char buffer[64];
    if (getSensitiveData(buffer, sizeof(buffer))) {
        if (writeDB(key, buffer)) {
            // Record data
        }
    }
    (void) memset(buffer, 0, sizeof(buffer)); // Defect
}

void manageGBuffer(char *key) {
    //...
    memset(Gbuffer, 0, sizeof(Gbuffer)); // No Defect
}
```

In this example, the command `memset()` is used in `Job()` to clean the memory in `buffer`. Because `buffer` is not read after the cleaning, the compiler might remove this code when optimizing the code. Polyspace flags the uncertain cleaning. In the function `manage()`, `memset()` is used to clean the global array `Gbuffer`. Polyspace does not flag this cleaning operation.

#### Correction — Use Secure Functions to Clean Sensitive Memory

To fix this defect, use secure functions, such as `memset_s()`, to clean uncertain memory.

```
#include <string.h>

extern int getSensitiveData(char*, size_t);
extern int writeDB(char*, char*);

void Job(char *key) {
    char buffer[64];
    if (getSensitiveData(buffer, sizeof(buffer))) {
        if (writeDB(key, buffer)) {
            // Record data
        }
    }
    memset_s(buffer, 0, sizeof(buffer)); // No Defect
}
```

## Result Information

**Group:** Security
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `UNCERTAIN_MEMORY_CLEANING`
**Impact:** Medium

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

14

# Cryptography Defects

# Constant block cipher initialization vector

Initialization vector is constant instead of randomized

## Description

This defect occurs when you use a constant for the initialization vector (IV) during encryption.

### Risk

Using a constant IV is equivalent to not using an IV. Your encrypted data is vulnerable to dictionary attacks.

Block ciphers break your data into blocks of fixed size. Block cipher modes such as CBC (Cipher Block Chaining) protect against dictionary attacks by XOR-ing each block with the encrypted output from the previous block. To protect the first block, these modes use a random initialization vector (IV). If you use a constant IV to encrypt multiple data streams that have a common beginning, your data becomes vulnerable to dictionary attacks.

### Fix

Produce a random IV by using a strong random number generator.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

## Examples

### Constants Used for Initialization Vector

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

/* Using the cryptographic routines */

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16] = {'1', '2', '3', '4','5','6','b','8','9',
                                 '1','2','3','4','5','6','7'};
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the initialization vector `iv` has constants only. The constant initialization vector makes your cipher vulnerable to dictionary attacks.

#### Correction — Use Random Initialization Vector

One possible correction is to use a strong random number generator to produce the initialization vector. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

/* Using the cryptographic routines */

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_CIPHER_CONSTANT_IV`
**Impact:** Medium

# Version History

**Introduced in R2017a**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Constant cipher key

Encryption or decryption key is constant instead of randomized

## Description

This defect occurs when you use a constant for the encryption or decryption key.

### Risk

If you use a constant for the encryption or decryption key, an attacker can retrieve your key easily.

You use a key to encrypt and later decrypt your data. If a key is easily retrieved, data encrypted using that key is not secure.

### Fix

Produce a random key by using a strong random number generator.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

## Examples

### Constants Used for Key

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16] = {'1', '2', '3', '4','5','6','b','8','9',
                                 '1','2','3','4','5','6','7'};
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the cipher key, `key`, has constants only. An attacker can easily retrieve a constant key.

#### Correction — Use Random Key

Use a strong random number generator to produce the cipher key. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16
```

```
int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_CIPHER_CONSTANT_KEY`
**Impact:** Medium

## Version History

**Introduced in R2017a**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Context initialized incorrectly for cryptographic operation

Context used for public key cryptography operation is initialized for a different operation

## Description

This defect occurs when you initialize an EVP_PKEY_CTX object for a specific public key cryptography operation but use the object for a different operation.

For instance, you initialize the context for encryption.

```
ret = EVP_PKEY_encrypt_init(ctx);
```

However, you use the context for decryption without reinitializing the context.

```
ret = EVP_PKEY_decrypt(ctx, out, &out_len, in, in_len);
```

The checker detects if the context object used in these functions has been initialized by using the corresponding initialization functions: EVP_PKEY_paramgen, EVP_PKEY_keygen, EVP_PKEY_encrypt, EVP_PKEY_verify, EVP_PKEY_verify_recover,EVP_PKEY_decrypt, EVP_PKEY_sign, EVP_PKEY_derive,and EVP_PKEY_derive_set_peer.

### Risk

Mixing up different operations on the same context can lead to obscure code. It is difficult to determine at a glance whether the current object is used for encryption, decryption, signature, or another operation. The mixup can also lead to a failure in the operation or unexpected ciphertext.

### Fix

After you set up a context for a certain family of operations, use the context for only that family of operations.For instance, use these pairs of functions for initialization and usage of the EVP_PKEY_CTX context object.

- For encryption with EVP_PKEY_encrypt, initialize the context with EVP_PKEY_encrypt_init.
- For signature verification with EVP_PKEY_verify, initialize the context with EVP_PKEY_verify_init.
- For key generation with EVP_PKEY_keygen, initialize the context with EVP_PKEY_keygen_init.

If you want to reuse an existing context object for a different family of operations, reinitialize the context.

## Examples

### Encryption Using Context Initialized for Decryption

```
#include <openssl/evp.h>
```

```
#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf10;
size_t out_len10;
int func(unsigned char *src, size_t len, EVP_PKEY_CTX *ctx){
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_decrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf10, &out_len10, src, len);
}
```

In this example, the context is initialized for decryption but used for encryption.

**Correction — Use One Family of Operations**

One possible correction is to initialize the object for encryption.

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf10;
size_t out_len10;
int func(unsigned char *src, size_t len, EVP_PKEY_CTX *ctx){
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf10, &out_len10, src, len);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_PKEY_INCORRECT_INIT`
**Impact:** Medium

# Version History

**Introduced in R2018a**

## See Also

`Incorrect key for cryptographic algorithm` | `Missing data for encryption, decryption or signing operation` | `Missing parameters for key generation` | `Missing peer key` | `Missing private key` | `Missing public key` | `Nonsecure parameters for key generation` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"

“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Context initialized incorrectly for digest operation

Context used for digest operation is initialized for a different digest operation

## Description

This defect occurs when you initialize an `EVP_MD_CTX` context object for a specific digest operation but use the context for a different operation.

For instance, you initialize the context for creating a message digest only.

```
ret = EVP_DigestInit(ctx, EVP_sha256())
```

However, you perform a final step for signing:

```
ret = EVP_SignFinal(&ctx, out, &out_len, pkey);
```

The error is shown only if the final step is not consistent with the initialization of the context. If the intermediate update steps are inconsistent, it does not trigger an error because the intermediate steps do not depend on the nature of the operation. For instance, `EVP_DigestUpdate` works identically to `EVP_SignUpdate`.

### Risk

Mixing up different operations on the same context can lead to obscure code. It is difficult to determine at a glance whether the current object is used for message digest creation, signing, or verification. The mixup can also lead to a failure in the operation or unexpected message digest.

### Fix

After you set up a context for a certain family of operations, use the context for only that family of operations. For instance, use these pairs of functions for initialization and final steps.

- `EVP_DigestInit` : `EVP_DigestFinal`
- `EVP_DigestInit_ex` : `EVP_DigestFinal_ex`
- `EVP_DigestSignInit` : `EVP_DigestSignFinal`

If you want to reuse an existing context object for a different family of operations, reinitialize the context.

## Examples

### Inconsistent Initial and Final Digest Operation

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf16;
unsigned int out_len16;
```

```
void func(unsigned char *src, size_t len){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_sha256(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_SignUpdate(ctx, src, len);
  if (ret != 1) fatal_error();

  ret = EVP_DigestSignFinal(ctx, out_buf16, (size_t*) out_len16);

  if (ret != 1) fatal_error();
}
```

In this example, the context object is initialized for signing only with EVP_SignInit but the final step attempts to create a signed digest with EVP_DigestSignFinal.

#### Correction — Use One Family of Operations

One possible correction is to use the context object for signing only. Change the final step to EVP_SignFinal in keeping with the initialization step.

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf16;
unsigned int out_len16;

void corrected_cryptomdbadfunction(unsigned char *src, size_t len, EVP_PKEY* pkey){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_sha256(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_SignUpdate(ctx, src, len);
  if (ret != 1) fatal_error();

  ret = EVP_SignFinal(ctx, out_buf16, &out_len16, pkey);
  if (ret != 1) fatal_error();
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** CRYPTO_MD_BAD_FUNCTION
**Impact:** Medium

# Version History

**Introduced in R2018a**

## See Also

`Nonsecure hash algorithm` | `Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Incompatible padding for RSA algorithm operation

Cryptography operation is not supported by the padding type set in context

## Description

This defect occurs when you perform an RSA algorithm operation on a context object that is not compatible with the padding previously associated with the object.

For instance, you associate the OAEP padding scheme with a context object but later use the context for signature verification, an operation that the padding scheme does not support.

```
ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
...
ret = EVP_PKEY_verify(ctx, out, out_len, in, in_len);
```

### Risk

Padding schemes remove determinism from the RSA algorithm and protect RSA operations from certain kinds of attack.

When you use an incorrect padding scheme, the RSA operation can fail or result in unexpected ciphertext.

### Fix

Before performing an RSA operation, associate the context object with a padding scheme that is compatible with the operation.

- Encryption: Use the OAEP padding scheme.

  For instance, use the `EVP_PKEY_CTX_set_rsa_padding` function with the argument `RSA_PKCS1_OAEP_PADDING` or the `RSA_padding_add_PKCS1_OAEP` function.

  ```
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
  ```

  You can also use the PKCS#1v1.5 or SSLv23 schemes. Be aware that these schemes are considered insecure.

  You can then use functions such as `EVP_PKEY_encrypt` / `EVP_PKEY_decrypt` or `RSA_public_encrypt` / `RSA_private_decrypt` on the context.
- Signature: Use the RSA-PSS padding scheme.

  For instance, use the `EVP_PKEY_CTX_set_rsa_padding` function with the argument `RSA_PKCS1_PSS_PADDING`.

  ```
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PSS_PADDING);
  ```

  You can also use the ANSI X9.31, PKCS#1v1.5, or SSLv23 schemes. Be aware that these schemes are considered insecure.

  You can then use functions such as the `EVP_PKEY_sign`-`EVP_PKEY_verify` pair or the `RSA_private_encrypt`-`RSA_public_decrypt` pair on the context.

If you perform two kinds of operation with the same context, after the first operation, reset the padding scheme in the context before the second operation.

## Examples

### OAEP Padding for Signature Operation

```
#include <stddef.h>
#include <openssl/rsa.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;

int func(unsigned char *src, size_t len, RSA* rsa){
  if (rsa == NULL) fatal_error();
  return RSA_private_encrypt(len, src, out_buf, rsa, RSA_PKCS1_OAEP_PADDING);
}
```

In this example, the function `RSA_private_encrypt` performs a signature operation by using the OAEP padding scheme, which supports encryption operations only.

#### Correction — Use Padding Scheme That Supports Signature

One possible correction is to use the RSA-PSS padding scheme. The corrected example uses the function `RSA_padding_add_PKCS1_PSS` to associate the padding scheme with the context.

```
#include <stddef.h>
#include <openssl/evp.h>
#include <openssl/rsa.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *msg_pad;
unsigned char *out_buf;

int func(unsigned char *src, size_t len, RSA* rsa){
  if (rsa == NULL) fatal_error();

  ret = RSA_padding_add_PKCS1_PSS(rsa, msg_pad, src, EVP_sha256(), -2);
  if (ret <= 0) fatal_error();

  return RSA_private_encrypt(len, msg_pad, out_buf, rsa, RSA_NO_PADDING);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_RSA_BAD_PADDING`
**Impact:** Medium

## Version History

**Introduced in R2018a**

## See Also

`Missing blinding for RSA algorithm` | `Missing padding for RSA algorithm` | `Nonsecure RSA public exponent` | `Weak padding for RSA algorithm` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Inconsistent cipher operations

You perform encryption and decryption steps in succession with the same cipher context without a reinitialization in between

## Description

This defect occurs when you perform an encryption and decryption step with the same cipher context. You do not reinitialize the context in between those steps. The checker applies to symmetric encryption only.

For instance, you set up a cipher context for decryption using `EVP_DecryptInit_ex`.

```
EVP_DecryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);
```

However, you use the context for encryption using `EVP_EncryptUpdate`.

```
EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
```

### Risk

Mixing up encryption and decryption steps can lead to obscure code. It is difficult to determine at a glance whether the current cipher context is used for encryption or decryption. The mixup can also lead to race conditions, failed encryption, and unexpected ciphertext.

### Fix

After you set up a cipher context for a certain family of operations, use the context for only that family of operations.

For instance, if you set up a cipher context for decryption using `EVP_DecryptInit_ex`, use the context afterward for decryption only.

## Examples

### Encryption Step Following Decryption Step

```
#include <openssl/evp.h>
#include <stdlib.h>

/* Using the cryptographic routines */

unsigned char *out_buf;
int out_len;
unsigned char g_key[16];
unsigned char g_iv[16];
void func(unsigned char* src, int len) {

    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Cipher context set up for decryption*/
    EVP_DecryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, g_key, g_iv);

    /* Update step for encryption */
    EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

In this example, the cipher context `ctx` is set up for decryption using `EVP_DecryptInit_ex`. However, immediately afterward, the context is used for encryption using `EVP_EncryptUpdate`.

#### Correction — Change Setup Step

One possible correction is to change the setup step. If you want to use the cipher context for encryption, set it up using `EVP_EncryptInit_ex`.

```
#include <openssl/evp.h>
#include <stdlib.h>

unsigned char *out_buf;
int out_len;
unsigned char g_key[16];
unsigned char g_iv[16];

void func(unsigned char* src, int len) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Cipher context set up for encryption*/
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, g_key, g_iv);

    /* Update step for encryption */
    EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_CIPHER_BAD_FUNCTION`
**Impact:** Medium

## Version History

**Introduced in R2017a**

### See Also

`Find defects (-checkers)`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”

“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Incorrect key for cryptographic algorithm

Public key cryptography operation is not supported by the algorithm used in context initialization

## Description

This defect occurs when you initialize a context object with a key for a specific algorithm but perform an operation that the algorithm does not support.

For instance, you initialize the context with a key for the DSA algorithm.

```
ret = EVP_PKEY_set1_DSA(pkey,dsa);
ctx = EVP_PKEY_CTX_new(pkey, NULL);
```

However, you use the context for encrypting data, an operation that the DSA algorithm does not support.

```
ret = EVP_PKEY_encrypt(ctx,out, &out_len, in, in_len);
```

### Risk

If the algorithm does not support your cryptographic operation, you do not see the expected results. For instance, if you use the DSA algorithm for encryption, you might get unexpected ciphertext.

### Fix

Use the algorithm that is appropriate for the cryptographic operation that you want to perform:

- Diffie-Hellman (DH): For key derivation.
- Digital Signature Algorithm (DSA): For signature.
- RSA: For encryption and signature.
- Elliptic curve (EC): For key derivation and signature.

## Examples

### Encryption with DSA Algorithm

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(unsigned char *src, size_t len, DSA * dsa){
  EVP_PKEY_CTX *ctx;
  EVP_PKEY *pkey = NULL;

  pkey = EVP_PKEY_new();
  if(pkey == NULL) fatal_error();
  ret = EVP_PKEY_set1_DSA(pkey,dsa);
  if (ret <= 0) fatal_error();

  ctx = EVP_PKEY_CTX_new(pkey, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf, &out_len, src, len);
}
```

In this example, the context object is initialized with a key associated with the DSA algorithm. However, the object is used for encryption, an operation that the DSA algorithm does not support.

#### Correction — Use RSA Algorithm

One possible correction is to initialize the context object with a key associated with the RSA algorithm.

```
#include <openssl/evp.h>
#include <openssl/rsa.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(unsigned char *src, size_t len, RSA * rsa){
  EVP_PKEY_CTX *ctx;
  EVP_PKEY *pkey = NULL;

  pkey = EVP_PKEY_new();
  if(pkey == NULL) fatal_error();

  ret = EVP_PKEY_set1_RSA(pkey,rsa);
  if (ret <= 0) fatal_error();

  ctx = EVP_PKEY_CTX_new(pkey, NULL); /* RSA key is set in the context */
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx); /* Encryption operation is set in the context */
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf, &out_len, src, len);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_PKEY_INCORRECT_KEY`
**Impact:** Medium

## Version History

**Introduced in R2018a**

## See Also

`Context initialized incorrectly for cryptographic operation` | `Missing data for encryption, decryption or signing operation` | `Missing parameters for key generation` | `Missing peer key` | `Missing private key` | `Missing public key` | `Nonsecure parameters for key generation` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing blinding for RSA algorithm

Context used in decryption or signature verification is not blinded against timing attacks

## Description

This defect occurs when you do not enable blinding for an RSA context object before using the object for decryption or signature verification.

For instance, you do not turn on blinding in the context object `rsa` before this decryption step:

```
  ret = RSA_public_decrypt(in_len, in, out, rsa, RSA_PKCS1_PADDING)
```

### Risk

Without blinding, the time it takes for the cryptographic operation to be completed has a correlation with the key value. An attacker can gather information about the RSA key by measuring the time for completion. Blinding removes this correlation and protects the decryption or verification operation against timing attacks.

### Fix

Before performing RSA decryption or signature verification, enable blinding.

```
ret = RSA_blinding_on(rsa, NULL);
```

## Examples

### Blinding Disabled Before Decryption

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
int func(unsigned char *src, size_t len, RSA* rsa){
  if (rsa == NULL) fatal_error();

  RSA_blinding_off(rsa);
  return RSA_private_decrypt(len, src, out_buf, rsa, RSA_PKCS1_OAEP_PADDING);
}
```

In this example, blinding is disabled for the context object `rsa`. Decryption with this context object can be vulnerable to timing attacks.

#### Correction — Enable Blinding Before Decryption

One possible correction is to explicitly enable blinding before decryption. Even if blinding might be enabled previously or by default, explicitly enabling blinding ensures that the security of the current decryption step is not reliant on the caller of `func`.

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
int func(unsigned char *src, size_t len, RSA* rsa){
  if (rsa == NULL) fatal_error();

  ret = RSA_blinding_on(rsa, NULL);
  if (ret <= 0) fatal_error();
  return RSA_private_decrypt(len, src, out_buf, rsa, RSA_PKCS1_OAEP_PADDING);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_RSA_NO_BLINDING`
**Impact:** Medium

## Version History

**Introduced in R2018a**

### See Also

`Incompatible padding for RSA algorithm operation` | `Missing padding for RSA algorithm` | `Weak padding for RSA algorithm` | `Nonsecure RSA public exponent` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing block cipher initialization vector

Context used for encryption or decryption is associated with NULL initialization vector or not associated with an initialization vector

## Description

This defect occurs when you encrypt or decrypt data using a NULL initialization vector (IV).

**Note** You can initialize your cipher context with a NULL initialization vector (IV). However, if your algorithm requires an IV, before the encryption or decryption step, you must associate the cipher context with a non-NULL IV.

### Risk

Many block cipher modes use an initialization vector (IV) to prevent dictionary attacks. If you use a NULL IV, your encrypted data is vulnerable to such attacks.

Block ciphers break your data into blocks of fixed size. Block cipher modes such as CBC (Cipher Block Chaining) protect against dictionary attacks by XOR-ing each block with the encrypted output from the previous block. To protect the first block, these modes use a random initialization vector (IV). If you use a NULL IV, you get the same ciphertext when encrypting the same plaintext. Your data becomes vulnerable to dictionary attacks.

### Fix

Before your encryption or decryption steps

```
 ret = EVP_EncryptUpdate(&ctx, out_buf, &out_len, src, len)
```

associate your cipher context `ctx` with a non-NULL initialization vector.

```
ret = EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv)
```

## Examples

### NULL Initialization Vector Used for Encryption

```
#include <openssl/evp.h>
#include <stdlib.h>
#define fatal_error() abort()

unsigned char *out_buf;
int out_len;

int func(EVP_CIPHER_CTX *ctx, unsigned char *key, unsigned char *src, int len){
    if (key == NULL)
        fatal_error();
    /* Last argument is initialization vector */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, NULL);

    /* Update step with NULL initialization vector */
    return EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

In this example, the initialization vector associated with the cipher context `ctx` is NULL. If you use this context to encrypt your data, your data is vulnerable to dictionary attacks.

#### Correction — Use Random Initialization Vector

Use a strong random number generator to produce the initialization vector. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define fatal_error() abort()
#define SIZE16 16

unsigned char *out_buf;
int out_len;

int func(EVP_CIPHER_CTX *ctx, unsigned char *key, unsigned char *src, int len){
    if (key == NULL)
        fatal_error();
    unsigned char iv[SIZE16];
    RAND_bytes(iv, 16);

    /* Last argument is initialization vector */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Update step with non-NULL initialization vector */
    return EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_CIPHER_NO_IV`
**Impact:** Medium

# Version History

**Introduced in R2017a**

## See Also

`Find defects (-checkers)`

**Topics**

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Missing certification authority list

Certificate for authentication cannot be trusted

## Description

This defect occurs when you use a context to handle TLS/SSL connections with these functions, but you do not load a certification authority (CA) list into the context.

- `SSL_connect`
- `SSL_accept`
- `SSL_do_handshake`
- `SSL_write`
- `SSL_read`
- `BIO_do_connect`
- `BIO_do_accept`
- `BIO_do_handshake`

A CA is a trusted third party entity that issues digital certificates to other entities. The certificate contains information about its owner. Server or clients use this information to authenticate connections to the certificate owner.

The checker raises a defect if:

- For server authentication, the client has no CA list to determine whether the server certificate is from a trusted source.
- For client authentication, the server has no CA list to determine whether the client certificate is from a trusted source.

### Risk

Without a CA list, you cannot determine if the certificate is issued by a trusted CA. The entity that presents the certificate for authentication might not be the entity described in the certificate. Your connection is vulnerable to man-in-the-middle (MITM) attacks.

### Fix

Load a certification authority list into the context you create to handle TLS/SSL connections.

## Examples

### Missing CA List When `SSL_connect` Initiates TLS/SSL Handshake

```
#include <openssl/ssl.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <openssl/err.h>
```

```
unsigned char* buf;

int OpenConnection(char* hostname, int port)
{
    /* Open the connection */
}

SSL_CTX* InitCTX(void)
{
    SSL_CTX* ctx;
    OpenSSL_add_all_algorithms();
    ctx = SSL_CTX_new(TLSv1_2_client_method());
    if (ctx == NULL) {
        /*handle errors */
    }
    return ctx;
}

void func()
{
    SSL_CTX* ctx;
    int server;
    SSL* ssl;
    char buf[1024];
    int bytes;
    char* hostname, *portnum;
    int ret;

    SSL_library_init();
    hostname = "localhost";
    portnum = "4433";

    ctx = InitCTX();
    server = OpenConnection(hostname, atoi(portnum));
    ssl = SSL_new(ctx);
    SSL_set_fd(ssl, server);
    ret = SSL_connect(ssl);
    if (SSL_get_error(ssl, ret) <= 0) {
        char* msg = "Hello???";
        printf("Connected with %s encryption\n", SSL_get_cipher(ssl));
        SSL_write(ssl, msg, strlen(msg));
        bytes = SSL_read(ssl, buf, sizeof(buf));
        buf[bytes] = 0;
        printf("Received: \"%s\"\n", buf);
        SSL_free(ssl);
    } else
        ERR_print_errors_fp(stderr);
    close(server);
    SSL_CTX_free(ctx);
}
```

In this example, a context `ctx` is initialized to handle TLS/SSL connections. When `SSL_connect` initializes the TLS/SSL handshake with the server by using the SSL structure `ssl` created from `ctx`, there is no CA list to check the validity of the server certificate.

#### Correction — Before Initiating the TLS/SSL Handshake, Load a CA List into the Context

One possible correction is to, before you initialize the SSL structure, specify a list of CA certificates for the context `ctx`, for instance with `SSL_CTX_load_verify_locations`.

```
#include <openssl/ssl.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <openssl/err.h>

unsigned char* buf;

int OpenConnection(char* hostname, int port)
{
    /* Open the connection */
}

SSL_CTX* InitCTX(void)
{
    SSL_CTX* ctx;
    OpenSSL_add_all_algorithms();
    ctx = SSL_CTX_new(TLSv1_2_client_method());
    if (ctx == NULL) {
        /*handle errors */
    }
    return ctx;
}

void LoadCA(SSL_CTX* ctx, char* CertFile, char* CertPath)
{
    if (SSL_CTX_load_verify_locations(ctx, CertFile, CertPath) <= 0) {
        /* handle errors */
    }
}

void func()
{
    SSL_CTX* ctx;
    int server;
    SSL* ssl;
    char buf[1024];
    int bytes;
    char* hostname, *portnum;
    int ret;

    SSL_library_init();
    hostname = "localhost";
    portnum = "4433";

    ctx = InitCTX();
    LoadCA(ctx, "cacert.pem", "ca/");
    server = OpenConnection(hostname, atoi(portnum));
    ssl = SSL_new(ctx);
    SSL_set_fd(ssl, server);
    ret = SSL_connect(ssl);
    if (SSL_get_error(ssl, ret) <= 0) {
        char* msg = "Hello???";
        printf("Connected with %s encryption\n", SSL_get_cipher(ssl));
        SSL_write(ssl, msg, strlen(msg));
        bytes = SSL_read(ssl, buf, sizeof(buf));
        buf[bytes] = 0;
        printf("Received: \"%s\"\n", buf);
        SSL_free(ssl);
    } else
        ERR_print_errors_fp(stderr);
    close(server);
    SSL_CTX_free(ctx);
}
```

## Result Information

**Group:** Cryptography<br>
**Language:** C | C++<br>
**Default:** Off<br>
**Command-Line Syntax:** `CRYPTO_SSL_NO_CA`<br>
**Impact:** Medium

# Version History

**Introduced in R2019b**

## See Also

`Find defects (-checkers)` | `Missing X.509 certificate`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing cipher algorithm

An encryption or decryption algorithm is not associated with the cipher context

## Description

This defect occurs when you do not assign a cipher algorithm when setting up your cipher context.

You can initialize your cipher context without an algorithm. However, before you encrypt or decrypt your data, you must associate the cipher context with a cipher algorithm.

### Risk

A missing cipher algorithm can lead to run-time errors or at least, non-secure ciphertext.

Before encryption or decryption, you set up a cipher context that has the information required for encryption: the cipher algorithm and mode, an encryption or decryption key and an initialization vector (for modes that require initialization vectors).

```
ret = EVP_EncryptInit(&ctx, EVP_aes_128_cbc(), key, iv)
```

The function `EVP_aes_128_cbc()` specifies that the Advanced Encryption Standard (AES) algorithm must be used for encryption. The function also specifies a block size of 128 bits and the Cipher Bloch Chaining (CBC) mode.

Instead of specifying the algorithm, you can use NULL in the initialization step. However, before using the cipher context for encryption or decryption, you must perform an additional initialization that associates an algorithm with the context. Otherwise, the update steps for encryption or decryption can lead to run-time errors.

### Fix

Before your encryption or decryption steps

```
 ret = EVP_EncryptUpdate(&ctx, out_buf, &out_len, src, len)
```

associate your cipher context `ctx` with an algorithm.

```
ret = EVP_EncryptInit(ctx, EVP_aes_128_cbc(), key, iv)
```

## Examples

### Algorithm Missing During Context Initialization

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

unsigned char key[SIZE16];
unsigned char iv[SIZE16];
void func(void) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);
    EVP_EncryptInit_ex(ctx, NULL, NULL, key, iv);
}
```

In this example, an algorithm is not provided when the cipher context `ctx` is initialized.

Before you encrypt or decrypt your data, you have to provide a cipher algorithm. If you perform a second initialization to provide the algorithm, the cipher context is completely re-initialized. Therefore, the current initialization statement using `EVP_EncryptInit_ex` is redundant.

#### Correction — Provide Algorithm During Initialization

One possible correction is to provide an algorithm when you initialize the cipher context. In the corrected code below, the routine `EVP_aes_128_cbc` invokes the Advanced Encryption Standard (AES) algorithm. The routine also specifies a block size of 128 bits and the Cipher Block Chaining (CBC) mode for encryption.

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

unsigned char key[SIZE16];
unsigned char iv[SIZE16];
void func(unsigned char *src, int len, unsigned char *out_buf, int out_len) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Initialization of cipher context */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Update steps for encryption */
    EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_CIPHER_NO_ALGORITHM`
**Impact:** Medium

## Version History

**Introduced in R2017a**

### See Also

`Find defects (-checkers)`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”

“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Missing cipher data to process

Final encryption or decryption step is performed without previous update steps

## Description

This defect occurs when you perform the final step of a block cipher encryption or decryption incorrectly.

For instance, you do one of the following:

- You do not perform update steps for encrypting or decrypting the data before performing a final step.

```
/* Initialization of cipher context */
ret = EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);
...
/* Missing update step */
...
/* Final step */
ret = EVP_EncryptFinal_ex(ctx, out_buf, &out_len);
```

- You perform consecutive final steps without intermediate initialization and update steps.

```
/* Initialization of cipher context */
ret = EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);
...
/* Update step(s) */
ret = EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
...
/* Final step */
ret = EVP_EncryptFinal_ex(ctx, out_buf, &out_len);
...
/* Missing initialization and update */
...
/* Second final step */
ret = EVP_EncryptFinal_ex(ctx, out_buf, &out_len);
```

- You perform a cleanup of the cipher context and then perform a final step.

```
/* Initialization of cipher context */
ret = EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);
...
/* Update step(s) */
ret = EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
...
/* Cleanup of cipher context */
EVP_CIPHER_CTX_cleanup(ctx);
...
/* Second final step */
ret = EVP_EncryptFinal_ex(ctx, out_buf, &out_len);
```

### Risk

Block ciphers break your data into blocks of fixed size. During encryption or decryption, the update step encrypts or decrypts your data in blocks. Any leftover data is encrypted or decrypted by the final step. The final step adds padding to the leftover data so that it occupies one block, and then encrypts or decrypts the padded data.

If you perform the final step before performing the update steps, or perform the final step when there is no data to process, the behavior is undefined. You can also encounter run-time errors.

### Fix

Perform encryption or decryption in this sequence:

- Initialization of cipher context
- Update steps
- Final step
- Cleanup of context

## Examples

### Missing Update Steps for Encryption Before Final Step

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

unsigned char *out_buf;
int out_len;
unsigned char key[SIZE16];
unsigned char iv[SIZE16];

void func(void) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Initialization of cipher context */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Missing update steps for encryption */

    /* Final encryption step */
    EVP_EncryptFinal_ex(ctx, out_buf, &out_len);
}
```

In this example, after the cipher context is initialized, there are no update steps for encrypting the data. The update steps are supposed to encrypt one or more blocks of data, leaving the final step to encrypt data that is left over in a partial block. If you perform the final step without previous update steps, the behavior is undefined.

#### Correction — Perform Update Steps for Encryption Before Final Step

Perform update steps for encryption before the final step. In the corrected code below, the routine `EVP_EncryptUpdate` performs the update steps.

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

unsigned char *out_buf;
int out_len;
unsigned char key[SIZE16];
unsigned char iv[SIZE16];

void func(unsigned char *src, int len) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Initialization of cipher context */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Update steps for encryption */
    EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);

    /* Final encryption step */
    EVP_EncryptFinal_ex(ctx, out_buf, &out_len);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_CIPHER_NO_DATA`
**Impact:** Medium

## Version History

**Introduced in R2017a**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing cipher final step

You do not perform a final step after update steps for encrypting or decrypting data

## Description

This defect occurs when you do not perform a final step after your update steps for encrypting or decrypting data.

For instance, you do the following:

```
/* Initialization of cipher context */
ret = EVP_EncryptInit_ex(&ctx, EVP_aes_128_cbc(), NULL, key, iv);
...
/* Update step */
ret = EVP_EncryptUpdate(&ctx, out_buf, &out_len, src, len);
...
/* Missing final step */
...
/* Cleanup of cipher context */
EVP_CIPHER_CTX_cleanup(ctx);
```

### Risk

Block ciphers break your data into blocks of fixed size. During encryption or decryption, the update step encrypts or decrypts your data in blocks. Any leftover data is encrypted or decrypted by the final step. The final step adds padding to the leftover data so that it occupies one block, and then encrypts or decrypts the padded data.

If you do not perform the final step, leftover data remaining in a partial block is not encrypted or decrypted. You can face incomplete or unexpected output.

### Fix

After your update steps for encryption or decryption, perform a final step to encrypt or decrypt leftover data.

```
/* Initialization of cipher context */
ret = EVP_EncryptInit_ex(&ctx, EVP_aes_128_cbc(), NULL, key, iv);
...
/* Update step(s) */
ret = EVP_EncryptUpdate(&ctx, out_buf, &out_len, src, len);
...
/* Final step */
ret = EVP_EncryptFinal_ex(&ctx, out_buf, &out_len);
...
/* Cleanup of cipher context */
EVP_CIPHER_CTX_cleanup(ctx);
```

## Examples

### Cleanup of Cipher Context Before Final Step

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

unsigned char *out_buf;
int out_len;
unsigned char key[SIZE16];
unsigned char iv[SIZE16];

void func(unsigned char *src, int len) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Initialization of cipher context */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Update steps for encryption */
    EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);

    /* Missing final encryption step */

    /* Cleanup of cipher context */
    EVP_CIPHER_CTX_cleanup(ctx);
}
```

In this example, the cipher context `ctx` is cleaned up before a final encryption step. The final step is supposed to encrypt leftover data. Without the final step, the encryption is incomplete.

#### Correction — Perform Final Encryption Step

After your update steps for encryption, perform a final encryption step to encrypt leftover data. In the corrected code below, the routine `EVP_EncryptFinal_ex` is used to perform this final step.

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

unsigned char *out_buf;
int out_len;
unsigned char key[SIZE16];
unsigned char iv[SIZE16];

void func(unsigned char *src, int len) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);

    /* Initialization of cipher context */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Update steps for encryption */
    EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);

    /* Final encryption step */
    EVP_EncryptFinal_ex(ctx, out_buf, &out_len);

    /* Cleanup of cipher context */
    EVP_CIPHER_CTX_cleanup(ctx);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_CIPHER_NO_FINAL`
**Impact:** Medium

## Version History

**Introduced in R2017a**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing cipher key

Context used for encryption or decryption is associated with NULL key or not associated with a key

## Description

This defect occurs when you encrypt or decrypt data using a NULL encryption or decryption key.

**Note** You can initialize your cipher context with a NULL key. However, before you encrypt or decrypt your data, you must associate the cipher context with a non-NULL key.

### Risk

Encryption or decryption with a NULL key can lead to run-time errors or at least, non-secure ciphertext.

### Fix

Before your encryption or decryption steps

```
 ret = EVP_EncryptUpdate(&ctx, out_buf, &out_len, src, len)
```

associate your cipher context `ctx` with a non-NULL key.

```
ret = EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv)
```

Sometimes, you initialize your cipher context with a non-NULL key

```
ret = EVP_EncryptInit_ex(&ctx, cipher_algo_1, NULL, key, iv)
```

but change the cipher algorithm later. When you change the cipher algorithm, you use a NULL key.

```
 ret = EVP_EncryptInit_ex(&ctx, cipher_algo_2, NULL, NULL, NULL)
```

The second statement reinitializes the cipher context completely but with a NULL key. To avoid this issue, every time you initialize a cipher context with an algorithm, associate it with a key.

## Examples

### NULL Key Used for Encryption

```
#include <openssl/evp.h>
#include <stdlib.h>
#define fatal_error() abort()

unsigned char *out_buf;
int out_len;

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv, unsigned char *src, int len){
    if (iv == NULL)
        fatal_error();

    /* Fourth argument is cipher key */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, NULL, iv);

    /* Update step with NULL key */
    return EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

In this example, the cipher key associated with the context `ctx` is NULL. When you use this context to encrypt your data, you can encounter run-time errors.

**Correction — Use Random Cipher Key**

Use a strong random number generator to produce the cipher key. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define fatal_error() abort()
#define SIZE16 16

unsigned char *out_buf;
int out_len;

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv, unsigned char *src, int len){
    if (iv == NULL)
        fatal_error();
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);

    /* Fourth argument is cipher key */
    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    /* Update step with non-NULL cipher key */
    return EVP_EncryptUpdate(ctx, out_buf, &out_len, src, len);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_CIPHER_NO_KEY`
**Impact:** Medium

# Version History

**Introduced in R2017a**

## See Also

`Find defects (-checkers)`

**Topics**

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing data for encryption, decryption or signing operation

Data provided for public key cryptography operation is NULL or data length is zero

## Description

This defect occurs when the data provided for an encryption, decryption, signing, or authentication operation is NULL or the data length is zero.

For instance, you unintentionally provide a NULL value for `in` or a zero value for `in_len` in this decryption operation:

```
ret = EVP_PKEY_decrypt(ctx, out, &out_len, in, in_len);
```

Or, you provide a NULL value for `md` or `sig`, or a zero value for `md_len` or `sig_len` in this verification operation:

```
ret = EVP_PKEY_verify(ctx, md, mdlen, sig, siglen);
```

### Risk

With NULL data or zero length, the operation does not occur. The redundant operation often indicates a coding error.

### Fix

Check the placement of the encryption, decryption, or signing operation. If the operation is intended to happen, make sure that the data provided is non-NULL. Set the data length to a nonzero value.

## Examples

### Zero Data Length for Signing Operation

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY_CTX * ctx){
  if (ctx == NULL) fatal_error();
  unsigned char* sig = (unsigned char*) "0123456789";
  unsigned char* md = (unsigned char*) "0123456789";

  ret = EVP_PKEY_verify_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_signature_md(ctx, EVP_sha256());
  if (ret <= 0) fatal_error();
  return EVP_PKEY_verify(ctx, sig, 0, md, 0);
}
```

In this example, the data lengths (third and fifth arguments to `EVP_PKEY_verify`) are zero. The operation fails.

#### Correction — Use Nonzero Data Length

One possible correction is to use a nonzero length for the signature and the data believed to be signed.

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY_CTX * ctx){
  if (ctx == NULL) fatal_error();
  unsigned char* sig = (unsigned char*) "0123456789";
  unsigned char* md = (unsigned char*) "0123456789";

  ret = EVP_PKEY_verify_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_signature_md(ctx, EVP_sha256());
  if (ret <= 0) fatal_error();
  return EVP_PKEY_verify(ctx, sig, 10, md, 10);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_PKEY_NO_DATA`
**Impact:** Medium

# Version History

**Introduced in R2018a**

## See Also

`Context initialized incorrectly for cryptographic operation` | `Incorrect key for cryptographic algorithm` | `Missing parameters for key generation` | `Missing peer key` | `Missing private key` | `Missing public key` | `Nonsecure parameters for key generation` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing final step after hashing update operation

Hash is incomplete or non-secure

## Description

The defect occurs when, after an update operation on a message digest context, you do not perform a final step before you clean up or reinitialize the context.

When you use message digest functions, you typically initialize a message digest context and perform at least one update step to add data into the context. You then sign, verify, or retrieve the data in the context as a final step.

### Risk

A missing final step might indicate that the hash is incomplete or is non-secure.

### Fix

Perform a final step to sign, verify, or retrieve date from the message digest context before you clean up or reinitialize the context.

## Examples

### Missing Final Step Before Context Cleanup

```
#include <stdlib.h>
#include <openssl/evp.h>


void func(unsigned char* src, int len, EVP_PKEY* pkey)
{
    int ret;

    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);

    ret = EVP_DigestVerifyInit(&ctx, NULL, EVP_sha256(), NULL, pkey);
    if (ret != 1) handle_error();

    ret = EVP_DigestVerifyUpdate(&ctx, src, len);
    if (ret != 1) handle_error();

    EVP_MD_CTX_cleanup(&ctx);
}
```

In this example, a verification context `ctx` is initialized and updated with data. The context is then cleaned up without being verified in a final step. Typically, you create a verification context to validate a previously signed message. Without the final step the signature on the message cannot be validated.

#### Correction — Perform Final Step Before Context Cleanup

One possible correction is to perform a final step to verify the signature of the verification context before you clean up the context.

```
#include <stdlib.h>
#include <openssl/evp.h>

unsigned char out_buf[EVP_MAX_MD_SIZE];
unsigned int out_len;

void handle_error()
{
    exit(-1);
}


void func(unsigned char* src, int len, EVP_PKEY* pkey)
{
    int ret;

    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);

    ret = EVP_DigestVerifyInit(&ctx, NULL, EVP_sha256(), NULL, pkey);
    if (ret != 1) handle_error();

    ret = EVP_DigestVerifyUpdate(&ctx, src, len);
    if (ret != 1) handle_error();

    ret = EVP_DigestVerifyFinal(&ctx, out_buf, out_len);
    if (ret != 1) handle_error();

    EVP_MD_CTX_cleanup(&ctx);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_MD_NO_FINAL`
**Impact:** Medium

## Version History

**Introduced in R2020a**

### See Also

`Find defects (-checkers)` | `No data added into context` | `Nonsecure hash algorithm`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”

“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Missing hash algorithm

Context in EVP routine is initialized without a hash algorithm

## Description

This defect occurs when you use a message digest context in these EVP routines, but you initialize the context without specifying a hash algorithm.

- `EVP_DigestFinal`
- `EVP_DigestSignFinal`
- `EVP_SignFinal`
- `EVP_VerifyFinal`

### Risk

Using a message digest context that was initialized without an algorithm to perform a hashing operation might result in a run-time error. Even if the hashing operation is successful, the resulting digest is not secure.

### Fix

Specify a hash algorithm when you initial a message digest context that you use in an EVP routine.

## Examples

### Context Used in EVP Routine After Context Cleanup

```
#include <openssl/evp.h>

void func(unsigned char* src, int len)
{
    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);

    EVP_VerifyInit(&ctx, EVP_sha256());
    EVP_MD_CTX_cleanup(&ctx);
    EVP_VerifyUpdate(&ctx, src, len);
}
```

In this example, context `ctx` is initialized with secure hash algorithm SHA-256. But, `ctx` is cleaned up before it is used by `EVP_VerifyUpdate`. The clean up of `ctx` frees up its resources and reinitializes it without a hash algorithm. The hashing operation of `EVP_VerifyUpdate` might result in a run-time error.

#### Correction — Clean Up Context Only After You No Longer Need It

One possible correction is to clean up the digest context only after you no longer need it.

```
#include <openssl/evp.h>

void func(unsigned char* src, int len)
{
    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);

    EVP_VerifyInit(&ctx, EVP_sha256());
    EVP_VerifyUpdate(&ctx, src, len);
    EVP_MD_CTX_cleanup(&ctx);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_MD_NO_ALGORITHM`
**Impact:** Medium

## Version History

**Introduced in R2019b**

### See Also

`Find defects (-checkers)` | `Nonsecure hash algorithm`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing padding for RSA algorithm

Context used in encryption or signing operation is not associated with any padding

## Description

This defect occurs when you perform RSA encryption or signature by using a context object without associating the object with a padding scheme.

For instance, you perform encryption by using a context object that was initially not associated with a specific padding.

```
ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_NO_PADDING);
...
ret = EVP_PKEY_encrypt(ctx, out, &out_len, in, in_len)
```

### Risk

Padding schemes remove determinism from the RSA algorithm and protect RSA operations from certain kinds of attack. Padding ensures that a given message does not lead to the same ciphertext each time it is encrypted. Without padding, an attacker can launch chosen-plaintext attacks against the cryptosystem.

### Fix

Before performing an RSA operation, associate the context object with a padding scheme that is compatible with the operation.

- Encryption: Use the OAEP padding scheme.

  For instance, use the `EVP_PKEY_CTX_set_rsa_padding` function with the argument `RSA_PKCS1_OAEP_PADDING` or the `RSA_padding_add_PKCS1_OAEP` function.

  ```
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
  ```

  You can also use the PKCS#1v1.5 or SSLv23 schemes. Be aware that these schemes are considered insecure.

  You can then use functions such as `EVP_PKEY_encrypt` / `EVP_PKEY_decrypt` or `RSA_public_encrypt` / `RSA_private_decrypt` on the context.
- Signature: Use the RSA-PSS padding scheme.

  For instance, use the `EVP_PKEY_CTX_set_rsa_padding` function with the argument `RSA_PKCS1_PSS_PADDING`.

  ```
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PSS_PADDING);
  ```

  You can also use the ANSI X9.31, PKCS#1v1.5, or SSLv23 schemes. Be aware that these schemes are considered insecure.

  You can then use functions such as the `EVP_PKEY_sign`-`EVP_PKEY_verify` pair or the `RSA_private_encrypt`-`RSA_public_decrypt` pair on the context.

If you perform two kinds of operation with the same context, after the first operation, reset the padding scheme in the context before the second operation.

## Examples

### Encryption Without Padding

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;


int func(unsigned char *src, size_t len){
  EVP_PKEY_CTX *ctx;
  EVP_PKEY* pkey;

  /* Key generation */
  ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA,NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_keygen(ctx, &pkey);
  if (ret <= 0) fatal_error();

  /* Encryption */
  EVP_PKEY_CTX_free(ctx);
  ctx = EVP_PKEY_CTX_new(pkey,NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf, &out_len, src, len);
}
```

In this example, before encryption with `EVP_PKEY_encrypt`, a specific padding is not associated with the context object `ctx`.

#### Correction — Set Padding in Context Before Encryption

One possible correction is to set the OAEP padding scheme in the context.

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)
```

```
int ret;
unsigned char *out_buf;
size_t out_len;


int func(unsigned char *src, size_t len){
  EVP_PKEY_CTX *ctx;
  EVP_PKEY* pkey;

  /* Key generation */
  ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA,NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_keygen(ctx, &pkey);
  if (ret <= 0) fatal_error();

  /* Encryption */
  EVP_PKEY_CTX_free(ctx);
  ctx = EVP_PKEY_CTX_new(pkey,NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx);
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
  if (ret <= 0) fatal_error();
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf, &out_len, src, len);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** CRYPTO_RSA_NO_PADDING
**Impact:** Medium

## Version History

**Introduced in R2018a**

### See Also

Incompatible padding for RSA algorithm operation | Missing blinding for RSA algorithm | Nonsecure RSA public exponent | Weak padding for RSA algorithm | Find defects (-checkers)

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"

"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing parameters for key generation

Context used for key generation is associated with NULL parameters

## Description

This defect occurs when you perform a key generation step with a context object without first associating the object with required parameters.

For instance, you associate a `EVP_PKEY_CTX` context object with an empty `EVP_PKEY` object `params` before key generation :

```
EVP_PKEY * params = EVP_PKEY_new();
...
EVP_PKEY_CTX * ctx = EVP_PKEY_CTX_new(params, NULL);
...
EVP_PKEY_keygen(ctx, &pkey);
```

### Risk

Without appropriate parameters, the key generation step does not occur. The redundant operation often indicates a coding error.

### Fix

Check the placement of the key generation step. If the operation is intended, make sure that the parameters are set before key generation.

Certain algorithms use default parameters. For instance, if you specify the DSA algorithm when creating the `EVP_PKEY_CTX` object, a default key length of 1024 bits is used:

```
kctx = EVP_PKEY_CTX_new_id(EVP_PKEY_DSA, NULL);
```

Specifying the algorithm during context creation is sufficient to avoid this defect. Only if you use the Elliptic Curve (EC) algorithm, you must also specify the curve explicitly before key generation.

However, the default parameters can generate keys that are too weak for encryption. Weak parameters can trigger another defect. To change default parameters, use functions specific to the algorithm. For instance, to set parameters, you can use these functions:

- Diffie-Hellman (DH): Use `EVP_PKEY_CTX_set_dh_paramgen_prime_len` and `EVP_PKEY_CTX_set_dh_paramgen_generator`.
- Digital Signature Algorithm (DSA): Use `EVP_PKEY_CTX_set_dsa_paramgen_bits`.
- RSA: Use `EVP_PKEY_CTX_set_rsa_padding`, `EVP_PKEY_CTX_set_rsa_pss_saltlen`, `EVP_PKEY_CTX_set_rsa_rsa_keygen_bits`, and `EVP_PKEY_CTX_set_rsa_keygen_pubexp`.
- Elliptic curve (EC): Use `EVP_PKEY_CTX_set_ec_paramgen_curve_nid` and `EVP_PKEY_CTX_set_ec_param_enc`.

## Examples

### Empty Parameters During Key Generation

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  EVP_PKEY * params = EVP_PKEY_new();
  if (params == NULL) fatal_error();

  EVP_PKEY_CTX * ctx = EVP_PKEY_CTX_new(params, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

In this example, the context object `ctx` is associated with an empty parameter object `params`. The context object does not have the required parameters for key generation.

#### Correction — Specify Algorithm During Context Creation

One possible correction is to specify an algorithm, such as RSA, during context creation. For stronger encryption, use 2048 bits for key length instead of the default 1024 bits.

```
#include <openssl/evp.h>
#include <openssl/rsa.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  EVP_PKEY_CTX * ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();

  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();

  return EVP_PKEY_keygen(ctx, &pkey);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_PKEY_NO_PARAMS`
**Impact:** Medium

## Version History

**Introduced in R2018a**

## See Also

`Context initialized incorrectly for cryptographic operation` | `Incorrect key for cryptographic algorithm` | `Missing data for encryption, decryption or signing` | `Missing peer key` | `Missing private key` | `Missing public key` | `Nonsecure parameters for key generation` | `Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Missing peer key

Context used for shared secret derivation is associated with NULL peer key or not associated with a peer key at all

## Description

This defect occurs when you use a context object for shared secret derivation but you have not previously associated the object with a non-NULL peer key.

For instance, you initialize the context object, and then use the object for shared secret derivation without an intermediate step where the object is associated with a peer key:

```
EVP_PKEY_derive_init(ctx);
/* Missing step for associating peer key with context */
ret = EVP_PKEY_derive(ctx, out_buf, &out_len);
```

The counterpart checker `Missing private key` checks for a private key in shared secret derivation.

### Risk

Without a peer key, the shared secret derivation step does not occur. The redundant operation often indicates a coding error.

### Fix

Check the placement of the shared secret derivation step. If the operation is intended, make sure that you have completed these steps prior to the operation:

- Generate a non-NULL peer key.

  For instance:

  ```
  EVP_PKEY* peerkey = NULL;
  EVP_PKEY_keygen(EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL), &peerkey);
  ```
- Associate a non-NULL context object with the peer key.

  For instance:

  ```
  EVP_PKEY_derive_set_peer(ctx,peerkey);
  ```

## Examples

### Missing Step for Associating Peer Key with Context

```
#include <stddef.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)
```

```
int ret;
unsigned char *out_buf;
size_t out_len;

int func(EVP_PKEY *pkey){
  if (pkey == NULL) fatal_error();

  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(pkey, NULL);
  if (ctx == NULL) fatal_error();
  ret = EVP_PKEY_derive_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_derive(ctx, out_buf, &out_len);
}
```

In this example, the context object `ctx` is associated with a private key but not a peer key. The `EVP_PKEY_derive` function uses this context object for shared secret derivation.

#### Correction — Set Peer Key in Context

One possible correction is to use the function `EVP_PKEY_derive_set_peer` and associate a peer key with the context object. Make sure that the peer key is non-NULL.

```
#include <stddef.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(EVP_PKEY *pkey,  EVP_PKEY* peerkey){
  if (pkey == NULL) fatal_error();
  if (peerkey == NULL) fatal_error();

  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(pkey, NULL);
  if (ctx == NULL) fatal_error();
  ret = EVP_PKEY_derive_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_derive_set_peer(ctx,peerkey);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_derive(ctx, out_buf, &out_len);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_PKEY_NO_PEER`
**Impact:** Medium

# Version History

**Introduced in R2018a**

## See Also

`Context initialized incorrectly for cryptographic operation` | `Incorrect key for cryptographic algorithm` | `Missing data for encryption, decryption or signing` | `Missing parameters for key generation` | `Missing private key` | `Missing public key` | `Nonsecure parameters for key generation` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing private key

Context used for cryptography operation is associated with NULL private key or not associated with a private key at all

## Description

This defect occurs when you use a context object for decryption, signature, or shared secret derivation but you have not previously associated the object with a non-NULL private key.

For instance, you initialize the context object with a NULL private key and use the object for decryption later.

```
ctx = EVP_PKEY_CTX_new(pkey, NULL);
...
ret = EVP_PKEY_decrypt_init(ctx);
...
ret = EVP_PKEY_decrypt(ctx, out, &out_len, in, in_len);
```

The counterpart checker `Missing public key` checks for a public key in encryption and authentication operations. The checker `Missing peer key` checks for a peer key in shared secret derivation.

### Risk

Without a private key, the decryption, signature, or shared secret derivation step does not occur. The redundant operation often indicates a coding error.

### Fix

Check the placement of the operation (decryption, signature, or shared secret derivation). If the operation is intended, make sure you have completed these steps prior to the operation:

- Generate a non-NULL private key.

  For instance:

  ```
  EVP_PKEY *pkey = NULL;
  kctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);

  EVP_PKEY_keygen_init(kctx);
  EVP_PKEY_CTX_set_rsa_keygen_bits(kctx, RSA_2048BITS);
  EVP_PKEY_keygen(kctx, &pkey);
  ```
- Associate a non-NULL context object with the private key.

  For instance:

  ```
  ctx = EVP_PKEY_CTX_new(pkey, NULL);
  ```

  Note: If you use `EVP_PKEY_CTX_new_id` instead of `EVP_PKEY_CTX_new`, you are not associating the context object with a private key.

## Examples

### Missing Step for Associating Private Key with Context

```
#include <stddef.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(unsigned char *src, size_t len){
  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_decrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_decrypt(ctx, out_buf, &out_len, src, len);
}
```

In this example, the context object ctx is initialized with EVP_PKEY_CTX_new_id instead of EVP_PKEY_CTX_new. The function EVP_PKEY_CTX_new_id does not associate the context object with a key. However, the EVP_PKEY_decrypt function uses this object for decryption.

#### Correction — Associate Private Key with Context During Initialization

One possible correction is to use the EVP_PKEY_CTX_new function for context initialization and associate a private key with the context object. In the following correction, the private key pkey is obtained from an external source and checked for NULL before use.

```
#include <stddef.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(unsigned char *src, size_t len, EVP_PKEY *pkey){
  if (pkey == NULL) fatal_error();

  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(pkey, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_decrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_decrypt(ctx, out_buf, &out_len, src, len);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_PKEY_NO_PRIVATE_KEY`
**Impact:** Medium

# Version History

**Introduced in R2018a**

## See Also

`Context initialized incorrectly for cryptographic operation` | `Incorrect key for cryptographic algorithm` | `Missing data for encryption, decryption or signing` | `Missing parameters for key generation` | `Missing peer key` | `Missing public key` | `Nonsecure parameters for key generation` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing private key for X.509 certificate

Missing key might result in run-time error or non-secure encryption

## Description

The defect occurs when you load a X.509 certificate file into the SSL context but you do not load the corresponding private key, or the key that you load into the context is null.

Typically, in a TLS/SSL exchange, the server proves its identity during a TLS/SSL handshake by sending a X.509 certificate that contains information about the server and a public key. The client that receives the certificate uses the public key to encrypt and send a pre-master secret that can only be decrypted with the corresponding private key. The server uses the decrypted pre-master secret and other exchanged messages to generate session keys that are used to encrypt the communication session.

The checker raises no defect if:

- You pass the SSL context as an argument to the function that calls `SSL_new`.
- You declare the SSL context outside the scope of the function handling the connection.

### Risk

Not loading the private key for a X.509 certificate might result in a run-time error on non-secure encryption.

### Fix

Load the private key of the X.509 certificate into the SSL context by calling `SSL_CTX_use_PrivateKey_file` or load the private key into the SSL structure by calling `SSL_use_PrivateKey_file`.

## Examples

### No Private Key Loaded Into SSL Context

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>
#define SSL_SERVER_CRT "server.pem"

#define fatal_error() exit(-1)

void load_cert(SSL_CTX* ctx, const char* certfile)
{
    int ret = SSL_CTX_use_certificate_file(ctx, certfile, SSL_FILETYPE_PEM);
    if (ret <= 0) fatal_error();
}

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;

    /* creation context for the SSL protocol */
    ctx = SSL_CTX_new(SSLv23_server_method());
    if (ctx == NULL) fatal_error();
    /* context configuration */
    load_cert(ctx, SSL_SERVER_CRT);

    /* Handle connection */
    ssl = SSL_new(ctx);
    ret = SSL_accept(ssl);
    if (ret <= 0) fatal_error();

    SSL_free(ssl);
    SSL_CTX_free(ctx);
}
```

In this example, SSL context `ctx` is initiated with server role and the function `load_cert` loads the server certificate into `ctx`. The server then waits for a client to initiate a handshake. However, since the private key is not loaded into the SSL structure, the server cannot decrypt the pre-master secret that a client sends, and the handshake fails.

#### Correction — Load Private Key into SSL Context

One possible correction is to load the private key into the SSL context after you load the server certificate file.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>
#define SSL_SERVER_CRT "server.pem"
#define SSL_SERVER_KEY "server.key"

#define fatal_error() exit(-1)

void load_cert(SSL_CTX* ctx, const char* certfile)
{
    int ret = SSL_CTX_use_certificate_file(ctx, certfile, SSL_FILETYPE_PEM);
    if (ret <= 0) fatal_error();

    ret = SSL_CTX_use_PrivateKey_file(ctx, SSL_SERVER_KEY, SSL_FILETYPE_PEM);
    if (ret <= 0) fatal_error();
}

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;

    /* creation context for the SSL protocol */
    ctx = SSL_CTX_new(SSLv23_server_method());
    if (ctx == NULL) fatal_error();

    /* context configuration */
    load_cert(ctx, SSL_SERVER_CRT);

    /* Handle connection */
    ssl = SSL_new(ctx);
    ret = SSL_accept(ssl);
    if (ret <= 0) fatal_error();

    SSL_free(ssl);
    SSL_CTX_free(ctx);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_SSL_NO_PRIVATE_KEY`
**Impact:** Medium

## Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)` | `Missing certification authority list` | `Missing X.509 certificate`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing public key

Context used for cryptography operation is associated with NULL public key or not associated with a public key at all

## Description

This defect occurs when you use a context object for encryption or signature authentication but you have not previously associated the object with a non-NULL public key.

For instance, you initialize the context object with a NULL public key and use the object for encryption later.

```
ctx = EVP_PKEY_CTX_new(pkey, NULL);
...
ret = EVP_PKEY_encrypt_init(ctx);
...
ret = EVP_PKEY_encrypt(ctx, out, &out_len, in, in_len);
```

The counterpart checker `Missing private key` checks for a private key in decryption and signature operations.

### Risk

Without a public key, the encryption or signature authentication step does not happen. The redundant operation often indicates a coding error.

### Fix

Check the placement of the operation (encryption or signature authentication). If the operation is intended to happen, make sure you have done these steps prior to the operation:

- You generated a non-NULL public key.

  For instance:

  ```
  EVP_PKEY *pkey = NULL;
  kctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);

  EVP_PKEY_keygen_init(kctx);
  EVP_PKEY_CTX_set_rsa_keygen_bits(kctx, RSA_2048BITS);
  EVP_PKEY_keygen(kctx, &pkey);
  ```
- You associated a non-NULL context object with the public key.

  For instance:

  ```
  ctx = EVP_PKEY_CTX_new(pkey, NULL);
  ```

  Note: If you use `EVP_PKEY_CTX_new_id` instead of `EVP_PKEY_CTX_new`, you are not associating the context object with a public key.

## Examples

### Missing Step for Associating Private Key with Context

```
#include <stddef.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(unsigned char *src, size_t len){
  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf, &out_len, src, len);
}
```

In this example, the context object ctx is initialized with EVP_PKEY_CTX_new_id instead of EVP_PKEY_CTX_new. The function EVP_PKEY_CTX_new_id does not associate the context object with a key. However, the EVP_PKEY_encrypt function uses this object for decryption.

**Correction — Associate Public Key with Context During Initialization**

One possible correction is to use the EVP_PKEY_CTX_new function for context initialization and associate a public key with the context object. In the following correction, the public key pkey is obtained from an external source and checked for NULL before use.

```
#include <stddef.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
size_t out_len;

int func(unsigned char *src, size_t len, EVP_PKEY *pkey){
  if (pkey == NULL) fatal_error();

  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(pkey, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_encrypt_init(ctx);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_encrypt(ctx, out_buf, &out_len, src, len);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_PKEY_NO_PUBLIC_KEY`
**Impact:** Medium

# Version History

**Introduced in R2018a**

## See Also

`Context initialized incorrectly for cryptographic operation` | `Incorrect key for cryptographic algorithm` | `Missing data for encryption, decryption or signing` | `Missing parameters for key generation` | `Missing peer key` | `Missing private key` | `Nonsecure parameters for key generation` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing salt for hashing operation

Hashed data is vulnerable to rainbow table attack

## Description

This defect occurs when you use a digest context in these functions, but you hash data into the context only once or you use a null salt in all subsequent hashing operations. A salt is random data that you use to improve the security of a hashing operation. The hashing operation takes the salt as an input to produce a more secure hashed value.

- `EVP_DigestFinal`
- `EVP_DigestSignUpdate`
- `EVP_DigestVerifyUpdate`
- `SHA*_Final` family of functions

**Missing salt for hashing operation** raises no defect if no information is available about the context. For instance, if the context is passed as an argument to the function that calls the hashing operation or if the context is declared outside the scope of the function. For example, no defect is raised in this code snippet.

```
EVP_MD_CTX ctx_global;

void foo(EVP_MD_CTX* ctx) {
//ctx passed as argument of func()
    EVP_DigestFinal(ctx, out_buf, &out_len); //no defect
}

void bar() {
// ctx_global declared outside of bar()
    EVP_DigestFinal(&ctx_glob, out_buf, &out_len); //no defect
}
```

### Risk

Hashing the same data without a salt results in the same hashed value. For instance, if you hash user passwords and two users have the same passwords, the hashed passwords are identical. The hashing is then vulnerable to precomputed rainbow attacks.

### Fix

Provide a salt when you hash data.

## Examples

### Data Hashed Into Context Only Once

```
#include <openssl/evp.h>
#include <cstring>
```

```
unsigned char* out_buf;
unsigned int out_len;

void func()
{
    const char* src = "toto";
    EVP_MD_CTX ctx;

    EVP_DigestInit(&ctx, EVP_sha256());
    EVP_DigestUpdate(&ctx, src, strlen(src));
    EVP_DigestFinal(&ctx, out_buf, &out_len);
    EVP_cleanup();
}
```

In this example, context `ctx` is initialized with secure hashing algorithm SHA-256, then `EVP_DigestUpdate` hashes `src` into `ctx`. Because `EVP_DigestUpdate` is called only once, no salt can be provided to improve the security of the hashing operation. The digest value that `EVP_DigestFinal` retrieves is then vulnerable to precomputed rainbow attacks.

#### Correction — Hash Salt Into Context After Initial Data Hash

One possible correction is to hash a salt into the context `ctx` after the first hashing operation. The resulting digest value that `EVP_DigestFinal` retrieves is more secure.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <cstring>

#define BUFF_SIZE_32 32

unsigned char* out_buf;
unsigned int out_len;

void func()
{
    const char* src = "toto";
    const char* salt;

    RAND_bytes((unsigned char*)salt, BUFF_SIZE_32);
    EVP_MD_CTX ctx;

    EVP_DigestInit(&ctx, EVP_sha256());
    EVP_DigestUpdate(&ctx, src, strlen(src));
    EVP_DigestUpdate(&ctx, salt, BUFF_SIZE_32);
    EVP_DigestFinal(&ctx, out_buf, &out_len);
    EVP_cleanup();
}
```

## Result Information

**Group:** Cryptography<br>
**Language:** C | C++<br>
**Default:** Off<br>
**Command-Line Syntax:** `CRYPTO_MD_NO_SALT`<br>
**Impact:** Medium

## Version History

**Introduced in R2019b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing X.509 certificate

Server or client cannot be authenticated

## Description

This defect occurs when you use a context to handle TLS/SSL connections with these functions, but you do not load an X.509 certificate into the context.

- `SSL_accept`
- `SSL_connect`
- `SSL_do_handshake`
- `SSL_write`
- `SSL_read`
- `BIO_do_accept`
- `BIO_do_connect`
- `BIO_do_handshake`

An X.509 certificate is a digital certificate that is issued to an entity. It contains information that identifies the entity. The certificate is used to authenticate connections to the entity identified in the certificate.

The checker raises a defect if:

- For a server authentication, no certificate is loaded before handling a connection.
- For a client authentication, the client attempts to connect to a server a second time after getting an `SSL_ERROR_WANT_X509_LOOKUP` error on the first connection attempt.

### Risk

When you do not load an X.509 certificate into the context to handle TLS/SSL connections, the connection is not secure and is vulnerable to man-in-the-middle (MITM) attacks.

### Fix

Load an X.509 certificate into the context you create to handle TLS/SSL connections.

## Examples

### SSL Structure Created From Context with Missing Certificate

```
#include <openssl/ssl.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <openssl/err.h>

unsigned char* buf;
int len;

SSL_CTX* InitServerCTX(void)
```

```
{
    SSL_CTX* ctx;
    OpenSSL_add_all_algorithms();
    ctx = SSL_CTX_new(SSLv23_server_method());
    SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2 | SSL_OP_NO_SSLv3 | SSL_OP_NO_TLSv1);
    if (ctx == NULL) {
        /*handle errors */
    }
    return ctx;
}

int OpenListener(int port)
{
    /* Create server socket */
}

void func()
{
    SSL_CTX* ctx;
    int server, port;
    int ret;
    SSL_library_init();

    ctx = InitServerCTX();
    server = OpenListener(port);
    while (1) {
        struct sockaddr_in addr;
        socklen_t len = sizeof(addr);
        SSL* ssl;

        int client = accept(server, (struct sockaddr*)&addr, &len);
        printf("Connection: %s:%d\n", inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
        ssl = SSL_new(ctx);
        SSL_set_fd(ssl, client);
        ret = SSL_accept(ssl);
        if (SSL_get_error(ssl, ret) <= 0)
            /* Serve connection */;
        else
            SSL_free(ssl);
    }
    close(server);
    SSL_CTX_free(ctx);
}
```

In this example, `InitServerCTX()` initializes context `ctx` for TLS/SSL connections, but no certificate is loaded into `ctx`. When `SSL_accept` checks the TLS/SLL handshake for the SSL structure created from `ctx`, there is no certificate available to authenticate the server.

### Correction — Before Creating a SSL Structure, Load Certificate Into Context

One possible correction is to, before you create a SSL structure, load a certificate into the context you create for TLS/SSL connections, for instance with `SSL_CTX_use_certificate_file`.

```
#include <openssl/ssl.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <openssl/err.h>

unsigned char* buf;
int len;

SSL_CTX* InitServerCTX(void)
{
    SSL_CTX* ctx;
    OpenSSL_add_all_algorithms();
    ctx = SSL_CTX_new(SSLv23_server_method());
    SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2 | SSL_OP_NO_SSLv3 | SSL_OP_NO_TLSv1);
    if (ctx == NULL) {
        /*handle errors */
    }
    return ctx;
}

void LoadCertificates(SSL_CTX* ctx, char* CertFile, char* KeyFile)
{
    if (SSL_CTX_use_certificate_file(ctx, CertFile, SSL_FILETYPE_PEM) <= 0) {
        /* Handle errors */
    }
}

int OpenListener(int port)
{
    /* Create server socket */
}

void func()
{
    SSL_CTX* ctx;
    int server, port;
    int ret;
    SSL_library_init();

    ctx = InitServerCTX();
    LoadCertificates(ctx, "mycert.pem", "mycert.pem");
    server = OpenListener(port);
    while (1) {
        struct sockaddr_in addr;
        socklen_t len = sizeof(addr);
        SSL* ssl;

        int client = accept(server, (struct sockaddr*)&addr, &len);
        printf("Connection: %s:%d\n", inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
        ssl = SSL_new(ctx);
        SSL_set_fd(ssl, client);
        ret = SSL_accept(ssl);
        if (SSL_get_error(ssl, ret) <= 0)
            /* Serve connection */;
        else
            SSL_free(ssl);
    }
    close(server);
    SSL_CTX_free(ctx);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** CRYPTO_SSL_NO_CERTIFICATE
**Impact:** Medium

# Version History

**Introduced in R2019b**

## See Also

Find defects (-checkers) | Missing certification authority list

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# No data added into context

Performing hash operation on empty context might cause run-time errors

## Description

The defect occurs when you only update a message digest context with null data, or you perform a final step on a message digest context without performing any update step.

When you use message digest functions, you typically initialize a message digest context and perform at least one update step to add data into the context. You then sign, verify, or retrieve the data in the context as a final step.

The checker raises no defect if no information is available about the context. For instance, if the context is passed as an argument to the function that calls the hashing operation or if the context is declared outside the scope of the function. For example, no defect is raised in this code snippet.

```
void bar(unsigned char* src, int len, EVP_MD_CTX *ctx) {
    //ctx passed as argument of bar()
    EVP_DigestFinal(ctx, out_buf, &out_len); //no defect
}
EVP_MD_CTX glob_ctx;
void foo(unsigned char* src, int len) {
    //glob_ctx declared outside scope of foo()
    EVP_DigestFinal(&glob_ctx, out_buf, &out_len); //no defect
}
```

### Risk

Performing an update step on a context with null data might result in a run-time error.

Performing a final step on a context with no data might result in unexpected behavior.

### Fix

Perform at least one update operation with non-null data on a message digest context before you sign, verify, or retrieve the data in the context.

## Examples

### No Update Step Before Final Step

```
#include <openssl/evp.h>
#include <stdio.h>

unsigned char out_buf[EVP_MAX_MD_SIZE];
unsigned int out_len;

void func(unsigned char* src, int len)
{
    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);
    EVP_DigestInit(&ctx, EVP_sha256());
    EVP_DigestUpdate(&ctx, src, len);
    EVP_MD_CTX_init(&ctx);
    EVP_DigestFinal(&ctx, out_buf, &out_len);
}
```

In this example, the message digest context `ctx` is initialized and an update operation is performed to add data `src` into the context. The context is then reinitialized but no data is added to `ctx` before `EVP_DigestFinal` attempts to retrieve data from `ctx`, which results in an error.

#### Correction — Perform Final Step Before Reinitializing Context

One possible correction is to perform the final step that retrieves data from the context before you reinitialize the context.

```
#include <openssl/evp.h>
#include <stdio.h>

unsigned char out_buf[EVP_MAX_MD_SIZE];
unsigned int out_len;

void func(unsigned char* src, int len)
{
    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);


    EVP_DigestInit(&ctx, EVP_sha256());
    EVP_DigestUpdate(&ctx, src, len);
    EVP_DigestFinal(&ctx, out_buf, &out_len);
    EVP_MD_CTX_init(&ctx);
}
```

### No Data Added to Context

```
#include <openssl/evp.h>
#include <stdio.h>

unsigned char out_buf[EVP_MAX_MD_SIZE];
unsigned int out_len;

void func(unsigned char* src, int len)
{
    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);
    size_t cnt = 0;

    EVP_DigestInit(&ctx, EVP_sha256());
    EVP_DigestUpdate(&ctx, src, cnt);
    EVP_DigestFinal(&ctx, out_buf, &out_len);
}
```

In this example, zero bytes of data is hashed into the message digest context during the update operation. Retrieving data from the context in the final step results in unexpected behavior.

#### Correction — Add non-Null Data Into Context

A possible correction is to add data into the context during the update step before you retrieve data from the context.

```
#include <openssl/evp.h>
#include <stdio.h>

unsigned char out_buf[EVP_MAX_MD_SIZE];
unsigned int out_len;

void func(unsigned char* src, int len)
{
    EVP_MD_CTX ctx;
    EVP_MD_CTX_init(&ctx);

    EVP_DigestInit(&ctx, EVP_sha256());
    EVP_DigestUpdate(&ctx, src, len);
    EVP_DigestFinal(&ctx, out_buf, &out_len);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_MD_NO_DATA`
**Impact:** Medium

## Version History

**Introduced in R2020a**

### See Also

`Find defects (-checkers)` | `Missing final step after hashing update operation`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Nonsecure hash algorithm

Context used for message digest creation is associated with weak algorithm

## Description

This defect occurs when you use a cryptographic hash function that is proven to be weak against certain forms of attack.

The hash functions flagged by this checker include SHA-0, SHA-1, MD4, MD5, and RIPEMD-160. The checker detects the use of these hash functions in:

- Functions from the EVP API such as `EVP_DigestUpdate` or `EVP_SignUpdate`.
- Functions from the low level API such as `SHA1_Update` or `MD5_Update`.

### Risk

You use a hash function to create a message digest from input data and thereby ensure integrity of your data. The hash functions flagged by this checker use algorithms with known weaknesses that an attacker can exploit. The attacks can comprise the integrity of your data.

### Fix

Use a more secure hash function. For instance, use the later SHA functions such as SHA-224, SHA-256, SHA-384, and SHA-512.

## Examples

### Use of MD5 Algorithm

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
unsigned int out_len;

void func(unsigned char *src, size_t len, EVP_PKEY* pkey){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_md5(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_DigestUpdate(ctx,src,len);

  if (ret != 1) fatal_error();

  ret = EVP_SignFinal(ctx, out_buf, &out_len, pkey);
  if (ret != 1) fatal_error();
}
```

In this example, during initialization with `EVP_SignInit_ex`, the context object is associated with the weak hash function MD5. The checker flags the usage of this context in the update step with `EVP_DigestUpdate`.

#### Correction — Use SHA-2 Family Function

One possible correction is to use a hash function from the SHA-2 family, such as SHA-256.

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
unsigned int out_len;

void func(unsigned char *src, size_t len, EVP_PKEY* pkey){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_sha256(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_SignUpdate(ctx, src, len);
  if (ret != 1) fatal_error();

  ret = EVP_SignFinal(ctx, out_buf, &out_len, pkey);
  if (ret != 1) fatal_error();
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_MD_WEAK_HASH`
**Impact:** Medium

# Version History

**Introduced in R2018a**

## See Also

`Context initialized incorrectly for digest operation` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Nonsecure parameters for key generation

Context used for key generation is associated with weak parameters

## Description

This defect occurs when you attempt key generation by using an EVP_PKEY_CTX context object that is associated with weak parameters. What constitutes a weak parameter depends on the public key algorithm used. In the DSA algorithm, a weak parameter can be the result of setting an insufficient parameter length.

For instance, you set the number of bits used for DSA parameter generation to 512 bits, and then use the parameters for key generation:

```
EVP_PKEY_CTX *pctx,*kctx;
EVP_PKEY *params, *pkey;

/* Initializations for parameter generation */
pctx = EVP_PKEY_CTX_new_id(EVP_PKEY_DSA, NULL);
params = EVP_PKEY_new();

/* Parameter generation */
ret = EVP_PKEY_paramgen_init(pctx);
ret = EVP_PKEY_CTX_set_dsa_paramgen_bits(pctx, KEYLEN_512BITS);
ret = EVP_PKEY_paramgen(pctx, &params);

/* Initializations for key generation */
kctx = EVP_PKEY_CTX_new(params, NULL);
pkey = EVP_PKEY_new();

/* Key generation */
ret = EVP_PKEY_keygen_init(kctx);
ret = EVP_PKEY_keygen(kctx, &pkey);
```

### Risk

Weak parameters lead to keys that are not sufficiently strong for encryption and expose sensitive information to known ways of attack.

### Fix

Depending on the algorithm, use these parameters:

- Diffie-Hellman (DH): Set the length of the DH prime parameter to 2048 bits.

  ```
  ret = EVP_PKEY_CTX_set_dh_paramgen_prime_len(pctx, 2048);
  ```

  Set the DH generator to 2 or 5.

  ```
  ret = EVP_PKEY_CTX_set_dh_paramgen_generator(pctx, 2);
  ```
- Digital Signature Algorithm (DSA): Set the number of bits used for DSA parameter generation to 2048 bits.

  ```
  ret = EVP_PKEY_CTX_set_dsa_paramgen_bits(pctx, 2048);
  ```

- RSA: Set the RSA key length to 2048 bits.

  ```
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(kctx, 2048);
  ```
- Elliptic curve (EC): Avoid using curves that are known to be broken, for instance, `X9_62_prime256v1`. Use, for instance, `sect239k1`.

  ```
  ret = EVP_PKEY_CTX_set_ec_paramgen_curve_nid(pctx, NID_sect239k1);
  ```

## Examples

### Insufficient Bits for RSA Key Generation

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  EVP_PKEY_CTX * ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 512);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

In this example, the RSA key generation uses 512 bits, which makes the generated key vulnerable to attacks.

**Correction — Use 2048 bits**

Use 2048 bits for RSA key generation.

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  EVP_PKEY_CTX * ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_PKEY_WEAK_PARAMS`
**Impact:** Medium

# Version History

**Introduced in R2018a**

## See Also

`Context initialized incorrectly for cryptographic operation` | `Incorrect key for cryptographic algorithm` | `Missing data for encryption, decryption or signing` | `Missing parameters for key generation` | `Missing peer key` | `Missing private key` | `Missing public key` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

### External Websites

https://safecurves.cr.yp.to/
https://csrc.nist.gov/publications/detail/fips/186/4/final

# Nonsecure RSA public exponent

Context used in key generation is associated with low exponent value

## Description

This defect occurs when you attempt RSA key generation by using a context object that is associated with a low public exponent.

For instance, you set a public exponent of 3 in the context object, and then use it for key generation.

```
/* Set public exponent */
ret = BN_dec2bn(&pubexp, "3");

/* Initialize context */
ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
pkey = EVP_PKEY_new();
ret = EVP_PKEY_keygen_init(kctx);

/* Set public exponent in context */
ret = EVP_PKEY_CTX_set_rsa_keygen_pubexp(ctx, pubexp);

/* Generate key */
ret = EVP_PKEY_keygen(kctx, &pkey);
```

### Risk

A low RSA public exponent makes certain kinds of attacks more damaging, especially when a weak padding scheme is used or padding is not used at all.

### Fix

It is recommended to use a public exponent of 65537. Using a higher public exponent can make the operations slower.

## Examples

### Using RSA Public Exponent of 3

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  BIGNUM* pubexp;
  EVP_PKEY_CTX* ctx;

  pubexp = BN_new();
  if (pubexp == NULL) fatal_error();
  ret = BN_set_word(pubexp, 3);
  if (ret <= 0) fatal_error();

  ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_pubexp(ctx, pubexp);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

In this example, an RSA public exponent of 3 is associated with the context object `ctx`. The low exponent makes operations that use the generated key vulnerable to certain attacks.

**Correction — Use Public Exponent of 65537**

One possible correction is to use the recommended public exponent 65537.

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  BIGNUM* pubexp;
  EVP_PKEY_CTX* ctx;

  pubexp = BN_new();
  if (pubexp == NULL) fatal_error();
  ret = BN_set_word(pubexp, 65537);
  if (ret <= 0) fatal_error();

  ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_pubexp(ctx, pubexp);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_RSA_LOW_EXPONENT`

**Impact:** Medium

## Version History

**Introduced in R2018a**

## See Also

`Incompatible padding for RSA algorithm operation` | `Missing padding for RSA algorithm` | `Missing blinding for RSA algorithm` | `Weak padding for RSA algorithm` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Nonsecure SSL/TLS protocol

Context used for handling SSL/TLS connections is associated with weak protocol

## Description

This defect occurs when you do not disable nonsecure protocols in an SSL_CTX or SSL context object before using the object for handling SSL/TLS connections.

For instance, you disable the protocols SSL2.0 and TLS1.0 but forget to disable the protocol SSL3.0, which is also considered weak.

```
/* Create and configure context */
ctx = SSL_CTX_new(SSLv23_method());
SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2|SSL_OP_NO_TLSv1);

/* Use context to handle connection */
ssl = SSL_new(ctx);
SSL_set_fd(ssl, NULL);
ret = SSL_connect(ssl);
```

### Risk

The protocols SSL2.0, SSL3.0, and TLS1.0 are considered weak in the cryptographic community. Using one of these protocols can expose your connections to cross-protocol attacks. The attacker can decrypt an RSA ciphertext without knowing the RSA private key.

### Fix

Disable the nonsecure protocols in the context object before using the object to handle connections.

```
/* Create and configure context */
ctx = SSL_CTX_new(SSLv23_method());
SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2|SSL_OP_NO_SSLv3|SSL_OP_NO_TLSv1);
```

## Examples

### Nonsecure Protocols Not Disabled

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>


#define fatal_error() exit(-1)

int ret;
int func(){
  SSL_CTX *ctx;
  SSL *ssl;

  SSL_library_init();

  /* context configuration */
  ctx = SSL_CTX_new(SSLv23_client_method());
  if (ctx==NULL) fatal_error();

  ret = SSL_CTX_use_certificate_file(ctx, "cert.pem", SSL_FILETYPE_PEM);
  if (ret <= 0) fatal_error();

  ret = SSL_CTX_load_verify_locations(ctx, NULL, "ca/path");
  if (ret <= 0) fatal_error();

  /* Handle connection */
  ssl = SSL_new(ctx);
  if (ssl==NULL) fatal_error();
  SSL_set_fd(ssl, NULL);

  return SSL_connect(ssl);
}
```

In this example, the protocols SSL2.0, SSL3.0, and TLS1.0 are not disabled in the context object before the object is used for a new connection.

**Correction — Disable Nonsecure Protocols**

Disable nonsecure protocols before using the objects for a new connection. Use the function `SSL_CTX_set_options` to disable the protocols SSL2.0, SSL3.0, and TLS1.0.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>


#define fatal_error() exit(-1)

int ret;
int func(){
  SSL_CTX *ctx;
  SSL *ssl;

  SSL_library_init();

  /* context configuration */
  ctx = SSL_CTX_new(SSLv23_client_method());
  if (ctx==NULL) fatal_error();

  SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2|SSL_OP_NO_SSLv3|SSL_OP_NO_TLSv1);

  ret = SSL_CTX_use_certificate_file(ctx, "cert.pem", SSL_FILETYPE_PEM);
  if (ret <= 0) fatal_error();
  ret = SSL_CTX_load_verify_locations(ctx, NULL, "ca/path");
  if (ret <= 0) fatal_error();

  /* Handle connection */
  ssl = SSL_new(ctx);
  if (ssl==NULL) fatal_error();
  SSL_set_fd(ssl, NULL);

  return SSL_connect(ssl);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_SSL_WEAK_PROTOCOL`
**Impact:** Medium

# Version History

**Introduced in R2018a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Predictable block cipher initialization vector

Initialization vector is generated from a weak random number generator

## Description

This defect occurs when you use a weak random number generator for the block cipher initialization vector.

### Risk

If you use a weak random number generator for the initiation vector, your data is vulnerable to dictionary attacks.

Block ciphers break your data into blocks of fixed size. Block cipher modes such as CBC (Cipher Block Chaining) protect against dictionary attacks by XOR-ing each block with the encrypted output from the previous block. To protect the first block, these modes use a random initialization vector (IV). If you use a weak random number generator for your IV, your data becomes vulnerable to dictionary attacks.

### Fix

Use a strong pseudo-random number generator (PRNG) for the initialization vector. For instance, use:

- OS-level PRNG such as `/dev/random` on UNIX or `CryptGenRandom()` on Windows
- Application-level PRNG such as Advanced Encryption Standard (AES) in Counter (CTR) mode, HMAC-SHA1, etc.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

## Examples

### Predictable Initialization Vector

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_pseudo_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the function `RAND_pseudo_bytes` declared in `openssl/rand.h` produces the initialization vector. The byte sequences that `RAND_pseudo_bytes` generates are not necessarily unpredictable.

#### Correction — Use Strong Random Number Generator

Use a strong random number generator to produce the initialization vector. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_CIPHER_PREDICTABLE_IV`
**Impact:** Medium

# Version History

**Introduced in R2017a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Predictable cipher key

Encryption or decryption key is generated from a weak random number generator

## Description

This defect occurs when you use a weak random number generator for the encryption or decryption key.

### Risk

If you use a weak random number generator for the encryption or decryption key, an attacker can retrieve your key easily.

You use a key to encrypt and later decrypt your data. If a key is easily retrieved, data encrypted using that key is not secure.

### Fix

Use a strong pseudo-random number generator (PRNG) for the key. For instance:

- Use an OS-level PRNG such as /dev/random on UNIX or CryptGenRandom() on Windows
- Use an application-level PRNG such as Advanced Encryption Standard (AES) in Counter (CTR) mode, HMAC-SHA1, etc.

For a list of random number generators that are cryptographically weak, see Vulnerable pseudo-random number generator.

## Examples

### Predictable Cipher Key

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_pseudo_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the function RAND_pseudo_bytes declared in openssl/rand.h produces the cipher key. However, the byte sequences that RAND_pseudo_bytes generates are not necessarily unpredictable.

### Correction — Use Strong Random Number Generator

One possible correction is to use a strong random number generator to produce the cipher key. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_CIPHER_PREDICTABLE_KEY`
**Impact:** Medium

# Version History

**Introduced in R2017a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Server certificate common name not checked

Attacker might use valid certificate to impersonate trusted host

## Description

The defect occurs when you do not check the common name provided in the server certificate against the domain name of the server.

Typically, when a client connects to a server, the server sends a digital certificate to the client that identifies the server as a trusted entity. The certificate contains information about the server, including the common name of the server. The common name matches the server domain name that the certificate identifies as a trusted entity.

The checker raises no defect if:

- You pass the SSL context as an argument to the function that calls `SSL_new`.
- You declare the SSL context outside the scope of the function handling the connection.

### Risk

A malicious attacker might use a valid certificate to impersonate a trusted host, resulting in the client interacting with an untrusted server.

### Fix

Use one of these functions to specify the server domain name that the program checks against the common name provided in the server certificate.

- `SSL_set_tlsext_host_name`
- `SSL_set1_host`
- `SSL_add1_host`

## Examples

### Client Checks Server Certificate but not Server Domain Name

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

void check_certificate(SSL_CTX* ctx, SSL* ssl)
{
    /* Check for Client authentication error */
    if (!SSL_get_peer_certificate(ssl)) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
    }
    /* Check for Client authentication error */
    if (SSL_get_verify_result(ssl) != X509_V_OK) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
    }
}

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;

    /* creation context for the SSL protocol */
    ctx = SSL_CTX_new(SSLv23_client_method());
    if (ctx == NULL) fatal_error();

    /* Handle connection */
    ssl = SSL_new(ctx);
    SSL_set_connect_state(ssl);
    check_certificate(ctx, ssl);
    ret = SSL_connect(ssl);
    if (ret <= 0) fatal_error();

    SSL_free(ssl);
    SSL_CTX_free(ctx);
}
```

In this example, an SSL structure is initiated with a client connection method. The client validates the server certificate with `check_certificate`. However, the client does not check that the certificate common name matches the domain name of the server. An attacker might use the valid certificate to impersonate the trusted server.

**Correction — Specify a Domain Name to Check Against the Certificate Common Name**

One possible correction is to use `SSL_set1_host` to specify the expected domain name that the program checks against the server certificate common name.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

void check_certificate(SSL_CTX* ctx, SSL* ssl)
{
    /* Check for Client authentication error */
    if (!SSL_get_peer_certificate(ssl)) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
    }
    /* Check for Client authentication error */
    if (SSL_get_verify_result(ssl) != X509_V_OK) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
    }
}

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;

    /* creation context for the SSL protocol */
    ctx = SSL_CTX_new(SSLv23_client_method());
    if (ctx == NULL) fatal_error();

    /* Handle connection */
    ssl = SSL_new(ctx);
    SSL_set_connect_state(ssl);
    check_certificate(ctx, ssl);
    ret = SSL_set1_host(ssl, "www.mysite.com");
    if (ret <= 0) fatal_error();
    ret = SSL_connect(ssl);
    if (ret <= 0) fatal_error();

    SSL_free(ssl);
    SSL_CTX_free(ctx);
}
```

## Result Information

**Group:** Cryptography<br>
**Language:** C | C++<br>
**Default:** Off<br>
**Command-Line Syntax:** `CRYPTO_SSL_HOSTNAME_NOT_CHECKED`<br>
**Impact:** Medium

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# TLS/SSL connection method not set

Program cannot determine whether to call client or server routines

## Description

The defect occurs when you call one of these functions without explicitly setting the connection method of the TLS/SSL context.

- SSL_read
- SSL_write
- SSL_do_handshake

The communication between server and client entities that use a TLS/SSL connection begins with a handshake. During the handshake, the parties exchange information and establish the encryption algorithm and session keys the parties use during the session. The connection methods for the server and client use different routines for the handshake.

The checker raises no defect if:

- You use SSL_connect (client) and SSL_accept (server) functions. These functions set the correct handshake routines automatically.
- You pass the SSL context as an argument to the function that calls SSL_new.
- You declare the SSL context outside the scope of the function handling the connection.

### Risk

You cannot begin a handshake if the SSL engine does not know which connection method routines to call.

### Fix

- For client handshake routines, call SSL_set_connect_state before you begin the handshake.
- For server handshake routines, call SSL_set_accept_state before you begin the handshake.

## Examples

### Server Connection Method Not Set Explicitly

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

int len;
unsigned char buf;
volatile int rd;
```

```
const SSL_METHOD*  set_method()
{
    return SSLv23_server_method();
}

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;
    const SSL_METHOD* method =  set_method();
    ctx = SSL_CTX_new(method);
    ssl = SSL_new(ctx);

    switch (rd) {
    case 1:
        ret = SSL_read(ssl, (void*)buf, len);
        if (ret <= 0) fatal_error();
        break;
    case 2:
        ret = SSL_do_handshake(ssl);
        if (ret <= 0) fatal_error();
        break;
    default:
        ret = SSL_write(ssl, (void*)buf, len);
        if (ret <= 0) fatal_error();
        break;
    }
}
```

In this example, the SSL context `ctx` is generated with server connection method `SSLv23_server_method`. However, the connection method is not set explicitly for the SSL structure `ssl` before the attempt to read from the connection, initiate a handshake, or write to the connection.

**Correction — Set Server Connection Method Explicitly**

One possible correction is to call `SSL_set_accept_state` to set the server role for the SSL structure `ssl` before you begin the handshake.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

int len;
unsigned char buf;
volatile int rd;

const SSL_METHOD*  set_method()
{
    return SSLv23_server_method();
}

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;
    const SSL_METHOD* method =  set_method();
    ctx = SSL_CTX_new(method);
    ssl = SSL_new(ctx);
    SSL_set_accept_state(ssl);


    switch (rd) {
    case 1:
        ret = SSL_read(ssl, (void*)buf, len);
        if (ret <= 0) fatal_error();
        break;
    case 2:
        ret = SSL_do_handshake(ssl);
        if (ret <= 0) fatal_error();
        break;
    default:
        ret = SSL_write(ssl, (void*)buf, len);
        if (ret <= 0) fatal_error();
        break;
    }
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** CRYPTO_SSL_NO_ROLE
**Impact:** Medium

# Version History

**Introduced in R2020a**

## See Also

Find defects (-checkers) | Missing X.509 certificate | Missing certification authority list

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# TLS/SSL connection method set incorrectly

Program calls functions that do not match role set by connection method

## Description

The defect occurs when you call functions that do not match the role set by the connection method that you specified for the SSL context.

The functions that you call when handling a TLS/SSL connection between client and server entities are different, depending on the role of the entity. For instance, the connection between a server and a client begins with a handshake. The client always initiates the handshake. You use `SSL_accept` with a server entity to wait for a client to initiate the handshake.

Typically, you set a connection method when you initiate the SSL context. The method specifies the role of the entity.

The checker flags the use of functions that do not match the connection method specified for the SSL context. See the **Event** column in the **Results Details** pane to view connection method specified for the SSL context.

### Risk

If you set the TLS/SSL connection method incorrectly, the functions you use to handle the connection might not match the role specified by the method. For instance, you use `SSL_accept` with a client entity to wait for a client to initiate a handshake instead of `SSL_connect` to initiate the handshake with a server.

### Fix

Make sure that you use functions that match the TLS/SSL connection method to handle the connection.

## Examples

### Client Waiting for Client to Initiate Handshake

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

const SSL_METHOD*  set_method()
{
    return SSLv23_client_method();
}

void set_method_1(SSL* ssl)
{
    SSL_set_connect_state(ssl);
}
void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;
    const SSL_METHOD* method = set_method();
    ctx = SSL_CTX_new(method);
    ssl = SSL_new(ctx);
    set_method_1(ssl);
    ret = SSL_accept(ssl);
    if (ret <= 0) fatal_error();
}
```

In this example, the SSL context `ctx` is initialized with a client role. The SSL structure is also explicitly set to client role through the call to `set_method_1`. To establish a connection with the server, the client should initiate a handshake with the server. Instead, `SSL_accept` causes the client to wait for another client to initiate a handshake.

#### Correction — Use SSL_connect to Initiate Handshake with Server

One possible correction is to use `SSL_connect` to initiate the TLS/SSL handshake with the server.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

const SSL_METHOD*  set_method()
{
    return SSLv23_client_method();
}

void set_method_1(SSL* ssl)
{
    SSL_set_connect_state(ssl);
}
void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;
    const SSL_METHOD* method = set_method();
    ctx = SSL_CTX_new(method);
    ssl = SSL_new(ctx);
    set_method_1(ssl);
    ret = SSL_connect(ssl);
    if (ret <= 0) fatal_error();
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off

**Command-Line Syntax:** `CRYPTO_SSL_BAD_ROLE`
**Impact:** Medium

## Version History

**Introduced in R2020a**

### See Also

`Find defects (-checkers)` | `Missing certification authority list` | `TLS/SSL connection method not set` | `Missing X.509 certificate`

**Topics**

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Weak cipher algorithm

Encryption algorithm associated with the cipher context is weak

## Description

This defect occurs when you associate a weak encryption algorithm with the cipher context.

### Risk

Some encryption algorithms have known flaws. Though the OpenSSL library still supports the algorithms, you must avoid using them.

If your cipher algorithm is weak, an attacker can decrypt your data by exploiting a known flaw or brute force attacks.

### Fix

Use algorithms that are well-studied and widely acknowledged as secure.

For instance, the Advanced Encryption Standard (AES) is a widely accepted cipher algorithm.

## Examples

### Use of DES Algorithm

```
#include <openssl/evp.h>
#include <stdlib.h>

void func(unsigned char *key, unsigned char *iv) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);
    const EVP_CIPHER * ciph = EVP_des_cbc();
    EVP_EncryptInit_ex(ctx, ciph, NULL, key, iv);
}
```

In this example, the routine `EVP_des_cbc()` invokes the Data Encryption Standard (DES) algorithm, which is now considered as non-secure and relatively slow.

#### Correction — Use AES Algorithm

One possible correction is to use the faster and more secure Advanced Encryption Standard (AES) algorithm instead.

```
#include <openssl/evp.h>
#include <stdlib.h>

void func(unsigned char *key, unsigned char *iv) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);
    const EVP_CIPHER * ciph = EVP_aes_128_cbc();
    EVP_EncryptInit_ex(ctx, ciph, NULL, key, iv);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_CIPHER_WEAK_CIPHER`
**Impact:** Medium

## Version History

**Introduced in R2017a**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Weak cipher mode

Encryption mode associated with the cipher context is weak

## Description

This defect occurs when you associate a weak block cipher mode with the cipher context.

The cipher mode that is especially flagged by this defect is the Electronic Code Book (ECB) mode.

### Risk

The ECB mode does not support protection against dictionary attacks.

An attacker can decrypt your data even using brute force attacks.

### Fix

Use a cipher mode more secure than ECB.

For instance, the Cipher Block Chaining (CBC) mode protects against dictionary attacks by:

- XOR-ing each block of data with the encrypted output from the previous block.
- XOR-ing the first block of data with a random initialization vector (IV).

## Examples

### Use of ECB Mode

```
#include <openssl/evp.h>
#include <stdlib.h>

void func(unsigned char *key, unsigned char *iv) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);
    const EVP_CIPHER * ciph = EVP_aes_128_ecb();
    EVP_EncryptInit_ex(ctx, ciph, NULL, key, iv);
}
```

In this example, the routine `EVP_aes_128_ecb()` invokes the Advanced Encryption Standard (AES) algorithm in the Electronic Code Book (ECB) mode. The ECB mode does not support protection against dictionary attacks.

#### Correction — Use CBC Mode

One possible correction is to use the Cipher Block Chaining (CBC) mode instead.

```
#include <openssl/evp.h>
#include <stdlib.h>
```

```
void func(unsigned char *key, unsigned char *iv) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);
    const EVP_CIPHER * ciph = EVP_aes_128_cbc();
    EVP_EncryptInit_ex(ctx, ciph, NULL, key, iv);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_CIPHER_WEAK_MODE`
**Impact:** Medium

## Version History

**Introduced in R2017a**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Weak padding for RSA algorithm

Context used in encryption or signing operation is associated with insecure padding type

## Description

This defect occurs when you perform RSA encryption or signature by using a context object that was previously associated with a weak padding scheme.

For instance, you perform encryption by using a context object that is associated with the PKCS#1v1.5 padding scheme. The scheme is considered insecure and has already been broken.

```
ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PADDING);
...
ret = EVP_PKEY_encrypt(ctx, out, &out_len, in, in_len)
```

### Risk

Padding schemes remove determinism from the RSA algorithm and protect RSA operations from certain kinds of attacks. Padding schemes such as PKCS#1v1.5, ANSI X9.31, and SSLv23 are known to be vulnerable. Do not use these padding schemes for encryption or signature operations.

### Fix

Before performing an RSA operation, associate the context object with a strong padding scheme.

- Encryption: Use the OAEP padding scheme.

  For instance, use the `EVP_PKEY_CTX_set_rsa_padding` function with the argument `RSA_PKCS1_OAEP_PADDING` or the `RSA_padding_add_PKCS1_OAEP` function.

  ```
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
  ```

  You can then use functions such as `EVP_PKEY_encrypt` / `EVP_PKEY_decrypt` or `RSA_public_encrypt` / `RSA_private_decrypt` on the context.
- Signature: Use the RSA-PSS padding scheme.

  For instance, use the `EVP_PKEY_CTX_set_rsa_padding` function with the argument `RSA_PKCS1_PSS_PADDING`.

  ```
  ret = EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PSS_PADDING);
  ```

  You can then use functions such as the `EVP_PKEY_sign`-`EVP_PKEY_verify` pair or the `RSA_private_encrypt`-`RSA_public_decrypt` pair on the context.

## Examples

### Encryption with PKCS#1v1.5 Padding

```
#include <stddef.h>
#include <openssl/rsa.h>
```

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;

int func(unsigned char *src, size_t len, RSA* rsa){
  if (rsa == NULL) fatal_error();

  return RSA_public_encrypt(len, src, out_buf, rsa, RSA_PKCS1_PADDING);
}
```

In this example, the PKCS#1v1.5 padding scheme is used in the encryption step.

#### Correction — Use OAEP Padding Scheme

Use the OAEP padding scheme for stronger encryption.

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;

int func(unsigned char *src, size_t len, RSA* rsa){
  if (rsa == NULL) fatal_error();

  return RSA_public_encrypt(len, src, out_buf, rsa, RSA_PKCS1_OAEP_PADDING);
}
```

### Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `CRYPTO_RSA_WEAK_PADDING`
**Impact:** Medium

## Version History

**Introduced in R2018a**

### See Also

`Incompatible padding for RSA algorithm operation` | `Missing padding for RSA algorithm` | `Missing blinding for RSA algorithm` | `Nonsecure RSA public exponent` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"

“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# X.509 peer certificate not checked

Connection might be vulnerable to man-in-the-middle attacks

## Description

The defect occurs when you do not properly check the X.509 certificate used to authenticate the TLS/SSL connection when handling the connection. To properly check the certificate, you must call these two functions together to obtain and verify the certificate.

- `SSL_get_peer_certificate`: Obtains a certificate from the client or server you are trying to authenticate. The function returns NULL if no certificate is present. Even if the function returns a certificate, the certificate must still be checked.
- `SSL_get_verify_result`: Verifies the certificate presented by the client or server. If you do not obtain a certificate before calling this function, there are no verification errors and the function returns successfully.

The checker raises a defect on the functions `SSL_read` or `SSL_write` when you attempt to read from or write to the TLS/SSL connection.

The checker raises no defect if:

- You declare the SSL context outside the scope of the function handling the connection.
- You use anonymous cypher suites.

### Risk

If you do not properly check the validity of the certificate of the peer you are attempting to authenticate, your connection is vulnerable to man-in-the-middle attacks.

### Fix

To properly check the validity of the certificate, call both `SSL_get_peer_certificate` and `SSL_get_verify_result`.

## Examples

### Client Certificate Obtained But Not Verified

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)

int len;
unsigned char buf;
```

```
void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;

    /* creation context for the SSL protocol */
    ctx = SSL_CTX_new(SSLv23_client_method());
    if (ctx == NULL) fatal_error();

    /* Set to require peer (client) certificate */
    SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);

    /* Handle connection */
    ssl = SSL_new(ctx);
    if (ssl == NULL) fatal_error();
    ret = SSL_set_fd(ssl, NULL);
    if (!ret) fatal_error();
    ret = SSL_connect(ssl);
    if (ret <= 0) fatal_error();

    /* Check for Client authentication error */
    if (!SSL_get_peer_certificate(ssl)) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
    }

    /*Read message from the client.*/
    ret = SSL_read(ssl, (void*)buf, len);
    if (ret <= 0) fatal_error();

    /* Close connection */
    SSL_free(ssl);
    SSL_CTX_free(ctx);
}
```

In this example, a TLS/SSL context is created for a server connection method. The function `SSL_get_peer_certificate` then requests the client certificate to authenticate the connection. However, the server then attempts to read from the connection without checking the validity of the returned certificate. The certificate might be invalid, and the connection could be vulnerable to a man-in-the-middle attack.

**Correction — Check the Validity of the Returned Certificate**

One possible correction is to check the validity of the returned certificate by calling `SSL_get_verify_result`.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

#define fatal_error() exit(-1)
```

```
int len;
unsigned char buf;

void func()
{
    int ret;
    SSL_CTX* ctx;
    SSL* ssl;

    /* creation context for the SSL protocol */
    ctx = SSL_CTX_new(SSLv23_client_method());
    if (ctx == NULL) fatal_error();

    /* Set to require peer (client) certificate */
    SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);

    /* Handle connection */
    ssl = SSL_new(ctx);
    if (ssl == NULL) fatal_error();
    ret = SSL_set_fd(ssl, NULL);
    if (!ret) fatal_error();
    ret = SSL_connect(ssl);
    if (ret <= 0) fatal_error();

    /* Check for Client authentication error */
    if (!SSL_get_peer_certificate(ssl)) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
    }

    if (SSL_get_verify_result(ssl) != X509_V_OK) {
        printf("SSL Client Authentication error\n");
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        exit(0);
    }

    /*Read message from the client.*/
    ret = SSL_read(ssl, (void*)buf, len);
    if (ret <= 0) fatal_error();

    /* Close connection */
    SSL_free(ssl);
    SSL_CTX_free(ctx);
}
```

## Result Information

**Group:** Cryptography
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** CRYPTO_SSL_CERT_NOT_CHECKED
**Impact:** Medium

## Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)` | `Missing certification authority list` | `Missing X.509 certificate`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

15

# Tainted Data Defects

# Array access with tainted index

Array index from unsecure source possibly outside array bounds

## Description

This defect occurs when you access an array by using an index that is obtained from unsecure sources and which has not been validated.

### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

### Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Use Index to Return Buffer Value

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];//Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The index `num` is obtained from an unsecure source and the function `taintedarrayindex` does not check to see if `num` is inside the range of `tab`.

### Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_ARRAY_INDEX`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Loop bounded with tainted value` | `Pointer dereference with tainted offset` | `Tainted size of variable length array` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Command executed from externally controlled path

Path argument from an unsecure source

## Description

This defect occurs when the path to a command executed in the program is constructed from external sources.

### Risk

An attacker can:

- Change the command that the program executes, possibly to a command that only the attack can control.
- Change the environment in which the command executes, by which the attacker controls what the command means and does.

### Fix

Before calling the command, validate the path to make sure that it is the intended location.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Executing Path from Environment Variable

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedpathcmd() {
    char cmd[SIZE128] = "";
    char* userpath = getenv("MYAPP_PATH");

    strncpy(cmd, userpath, SIZE100);//Noncompliant
    strcat(cmd, "/ls *");
    /* Launching command */
    system(cmd);//Noncompliant
}
```

This example obtains a path from an environment variable MYAPP_PATH. The path string is tainted. Polyspace flags its use in the strncopy function. system runs a command from the tainted path without checking the value of the path. If the path is not the intended path, your program executes in the wrong location.

**Correction — Use Trusted Path**

One possible correction is to use a list of allowed paths to match against the environment variable path.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

/* Function to sanitize a string */
/* Any defect is localized here */
int sanitize_str(char* s, size_t n) {
    int res = 0;
    /* String is ok if */
    if (s && n>0 && n<SIZE128) {
        /* - string is not null                     */
        /* - string has a positive and limited size */
        s[n-1] = '\0';  /* Add a security \0 char at end of string *///Noncompliant
        /* Tainted pointer detected above, used as "firewall" */
        res = 1;
    }
    return res;
}

/* Authorized path ids */
enum { PATH0=1, PATH1, PATH2 };

void taintedpathcmd() {
    char cmd[SIZE128] = "";

    char* userpathid = getenv("MYAPP_PATH_ID");
    if (sanitize_str(userpathid, SIZE100)) {
        int pathid = atoi(userpathid);


        char path[SIZE128] = "";
        switch(pathid) {
            case PATH0:
                strcpy(path, "/usr/local/my_app0");
                break;
            case PATH1:
                strcpy(path, "/usr/local/my_app1");
                break;
            case PATH2:
                strcpy(path, "/usr/local/my_app2");
                break;
            default:
                /* do nothing */
        break;
        }
        if (strlen(path)>0) {
            strncpy(cmd, path, SIZE100);
            strcat(cmd, "/ls *");
            system(cmd);
        }
    }
}
```

### Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_PATH_CMD`
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Execution of externally controlled command` | `Use of externally controlled environment variable` | `Host change using externally controlled elements` | `Library loaded from externally controlled path` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Execution of externally controlled command

Command argument from an unsecure source vulnerable to operating system command injection

## Description

This defect occurs when commands are fully or partially constructed from externally controlled input.

### Risk

Attackers can use the externally controlled input as operating system commands, or arguments to the application. An attacker could read or modify sensitive data can be read or modified, execute unintended code, or gain access to other aspects of the program.

### Fix

Validate the inputs to allow only intended input values. For example, create a list of acceptable inputs and compare the input against this list.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Call External Command

```
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "unistd.h"
#include "dlfcn.h"
#include "limits.h"
#define MAX 128


void taintedexternalcmd(void)
{
    char* usercmd;
    fgets(usercmd,MAX,stdin);
    char cmd[MAX] = "/usr/bin/cat ";
    strcat(cmd, usercmd);
    system(cmd);//Noncompliant
}
```

This example function calls a command from a user input without checking the command variable.

### Correction — Use a Predefined Command

One possible correction is to use a `switch` statement to run a predefined command, using the user input as the switch variable.

```
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "unistd.h"
#include "dlfcn.h"
#include "limits.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
enum { CMD0 = 1, CMD1, CMD2 };

void taintedexternalcmd(void)
{
    int usercmd = strtol(getenv("cmd"),NULL,10);
    char cmd[SIZE128] = "/usr/bin/cat ";

    switch(usercmd) {
        case CMD0:
            strcat(cmd, "*.c");
            break;
        case CMD1:
            strcat(cmd, "*.h");
            break;
        case CMD2:
            strcat(cmd, "*.cpp");
            break;
        default:
            strcat(cmd, "*.c");
    }
    system(cmd);
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_EXTERNAL_CMD`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Use of externally controlled environment variable` | `Host change using externally controlled elements` | `Command executed from externally controlled path` | `Library loaded from externally controlled path` | `Execution of a binary from a relative path can be controlled by an external actor` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Host change using externally controlled elements

Changing host ID from an unsecure source

## Description

This defect occurs when routines that change the host ID, such as `sethostid` (Linux) or `SetComputerName` (Windows), use arguments that are externally controlled.

### Risk

The tainted host ID value can allow external control of system settings. This control can disrupt services, cause unexpected application behavior, or cause other malicious intrusions.

### Fix

Use caution when changing or editing the host ID. Do not allow user-provided values to control sensitive data.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Change Host ID from Function Argument

```
#include <unistd.h>
#include <stdlib.h>

void bug_taintedhostid(void) {
    long userhid = strtol(getenv("HID"),NULL,10);
    sethostid(userhid);//Noncompliant
}
```

This example sets a new host ID using the argument passed to the function. Before using the host ID, check the value passed in.

#### Correction — Predefined Host ID

One possible correction is to change the host ID to a predefined ID. This example uses the host argument as a switch variable to choose between the different, predefined host IDs.

```
#include <unistd.h>
#include <stdlib.h>

extern long called_taintedhostid_sanitize(long);
enum { HI0 = 1, HI1, HI2, HI3 };
```

```
void taintedhostid(void) {
    long host = strtol(getenv("HID"),NULL,10);
    long hid = 0;
    switch(host) {
        case HI0:
            hid = 0x7f0100;
            break;
        case HI1:
            hid = 0x7f0101;
            break;
        case HI2:
            hid = 0x7f0102;
            break;
        case HI3:
            hid = 0x7f0103;
            break;
        default:
            /* do nothing */
        break;
    }
    if (hid > 0) {
        sethostid(hid);
    }
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_HOSTID`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Execution of externally controlled command` | `Use of externally controlled environment variable` | `Command executed from externally controlled path` | `Library loaded from externally controlled path` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Library loaded from externally controlled path

Using a library argument from an externally controlled path

## Description

This defect occurs when libraries are loaded from fixed or externally controlled unsecure paths and unintended actors can control one or more locations on the paths.

### Risk

If an attacker knows or controls the path that you use to load a library, the attacker can change:

- The library that the program loads, replacing the intended library and commands.
- The environment in which the library executes, giving unintended permissions and capabilities to the attacker.

### Fix

When possible, use hard-coded or fully qualified path names to load libraries. It is possible the hard-coded paths do not work on other systems. Use a centralized location for hard-coded paths, so that you can easily modify the path within the source code.

Another solution is to use functions that require explicit paths. For example, `system()` does not require a full path because it can use the `PATH` environment variable. However, `execl()` and `execv()` do require the full path.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Call Custom Library

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
```

```
void* taintedpathlib() {
    void* libhandle = NULL;
    char lib[SIZE128] = "";
    char* userpath = getenv("LD_LIBRARY_PATH");
    strncpy(lib, userpath, SIZE128);//Noncompliant- userpath is tainted
    strcat(lib, "/libX.so");
    libhandle = dlopen(lib, 0x00001);//Noncompliant
    return libhandle;
}
```

This example loads the library `libX.so` from an environment variable `LD_LIBRARY_PATH`. An attacker can change the library path in this environment variable. The actual library you load could be a different library from the one that you intend.

**Correction — Change and Check Path**

One possible correction is to change how you get the library path and check the path of the library before opening the library. This example receives the path as an input argument but then performs the following checks on the path:

- The function `sanitize_str` protects against possible buffer overflows.
- The function `identified_safe_libX_folder` checks if the path belongs to a list of allowed paths.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

/* Use allowlist */
static const char *libX_safe_folder[] = {
  "/usr/",
  "/usr/lib",
  "/lib"
};

/* Return the index if the input is in the allowlist */
int identified_safe_libX_folder(const char* path)
{
  for (int i = 0; i < sizeof(libX_safe_folder) / sizeof(libX_safe_folder[0]); i ++)
    {
      if (strcmp(path, libX_safe_folder[i]) == 0)
      return i;
    }
  return -1;
}

/* Function to sanitize a string */
```

```
char *sanitize_str(char* s, size_t n) {
  /* strlen is used here as a kind of firewall for tainted string errors */
  if (strlen(s) > 0 && strlen(s) < n)
    return s;
  else
    return NULL;
}

void* taintedpathlib(char* userpath) {
  void* libhandle = NULL;
  const char *const checked_userpath = sanitize_str(userpath, SIZE128);
  if (checked_userpath != NULL) {
    int index = identified_safe_libX_folder(checked_userpath);
    if (index > 0) {
      char lib[SIZE128] = "";
      strncpy(lib, libX_safe_folder[index], SIZE128);
      strcat(lib, "/libX.so");
      libhandle = dlopen(lib, RTLD_LAZY);
    }
  }
  return libhandle;
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_PATH_LIB`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Execution of externally controlled command` | `Use of externally controlled environment variable` | `Command executed from externally controlled path` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Loop bounded with tainted value

Loop controlled by a value from an unsecure source

## Description

This defect occurs when a loop is bounded by values obtained from unsecure sources.

### Risk

A tainted value can cause over looping or infinite loops. Attackers can use this vulnerability to terminate your program or cause other unintended behavior.

### Fix

Before starting the loop, validate unknown boundary and iterator values by validating their low bounds and high bounds. Execute the loop only when both the lower bound and upper bound of the tainted values are validated. Explicitly check that both the lower and upper bound of the tainted value is acceptable. Alternatively, saturate or clamp the tainted value.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Loop Boundary From User Input

```
#include<stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedloopboundary(void) {
    int count;
    scanf("%d", &count);
    int res = 0;
    for (int i=0 ; i < count; ++i) {//Noncompliant
        res += i;
    }
    return res;
}
```

In this example, the function uses a user input to loop `count` times. `count` could be any number because the value is not checked before starting the `for` loop.

#### Correction: Clamp Tainted Loop Control

One possible correction is to clamp the tainted loop control. To validate the tainted loop variable `count`, this example limits `count` to a minimum value and a maximum value by using inline functions `min` and `max`. Regardless of the user input, the value of `count` remains within a known range.

```
#include<stdio.h>
#include<algorithm>
#define MIN 50
#define MAX 128
static  inline int max(int a, int b) { return a > b ? a : b;}
static inline int min(int a, int b) { return a < b ? a : b; }

int taintedloopboundary(void) {
    int count;
    scanf("%d", &count);
    int res = 0;
    count = max(MIN, min(count, MAX));
    for (int i=0 ; i<count ; ++i) {
        res += i;
    }
    return res;
}
```

#### Correction — Check Tainted Loop Control

Another possible correction is to check the low bound and the high bound of the tainted loop boundary variable before starting the `for` loop. This example checks the low and high bounds of `count` and executes the loop only when `count` is between 0 and 127.

```
#include<stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};


int taintedloopboundary(void) {
    int count;
    scanf("%d", &count);
    int res = 0;

    if (count>=0 && count<SIZE128) {
        for (int i=0 ; i<count ; ++i) {
            res += i;
        }
    }
    return res;
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off

**Command-Line Syntax:** `TAINTED_LOOP_BOUNDARY`
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Array access with tainted index` | `Pointer dereference with tainted offset` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

**Topics**

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Memory allocation with tainted size

Size argument to memory function is from an unsecure source

## Description

This defect occurs when a memory allocation function, such as `calloc` or `malloc`, uses a size argument from an unsecure source.

### Risk

Uncontrolled memory allocation can cause your program to request too much system memory. This consequence can lead to a crash due to an out-of-memory condition, or assigning too many resources.

### Fix

Before allocating memory, check the value of your arguments to check that they do not exceed the bounds.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Allocate Memory Using Input From User

```
#include<stdio.h>
#include <stdlib.h>

int* bug_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = (int*)malloc(size);//Noncompliant
    return p;
}
```

In this example, `malloc` allocates `size` bytes of memory for the pointer `p`. The variable `size` comes from the user of the program. Its value is not checked, and it could be larger than the amount of available memory. If `size` is larger than the number of available bytes, your program could crash.

#### Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the `malloc` operation. This example checks to see if `size` is positive and less than the maximum size.

```
#include<stdio.h>
#include <stdlib.h>
```

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int* corrected_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = NULL;
    if (size>0 && size<SIZE128) {          /* Fix: Check entry range before use */
        p = (int*)malloc((unsigned int)size);
    }
    return p;
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_MEMORY_ALLOC_SIZE`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Unprotected dynamic memory allocation` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Pointer dereference with tainted offset

Offset is from an unsecure source and dereference might be out of bounds

## Description

This defect occurs when a pointer dereference uses an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

### Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Dereference Pointer Array

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset];//Noncompliant
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of `offset`. Continue with the pointer dereferencing only if `offset` is inside the valid range.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_PTR_OFFSET`
**Impact:** Low

## Version History

**Introduced in R2015b**

## See Also

`Array access with tainted index` | `Use of tainted pointer` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Tainted division operand

Operands of division operation (`/`) come from an unsecure source

## Description

This defect occurs when one or both integer operands in a division operation comes from unsecure sources.

### Risk

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

### Fix

Before performing the division, validate the values of the operands. Check for denominators of `0` or `-1`, and numerators of the minimum integer value.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See "Sources of Tainting in a Polyspace Analysis". To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Division of Function Arguments

```
#include <limits.h>
#include <stdio.h>

extern void print_int(int);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    int r =  num/denum; //Noncompliant
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

#### Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include <limits.h>
#include <stdio.h>

extern void print_long(long);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    long res= 0;
    if (denum!=0 && !(num==INT_MIN && denum==-1)) {
        res = num/denum;
    }
    print_long(res);
    return res;
}
```

### Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_INT_DIVISION`
**Impact:** Low

## Version History

**Introduced in R2015b**

### See Also

`Integer division by zero` | `Float division by zero` | `Tainted modulo operand` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Tainted modulo operand

Operands of remainder operation (%) come from an unsecure source

## Description

This defect occurs when one or both integer operands in a remainder operation (%) comes from unsecure sources.

### Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is -1, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

### Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of 0 and -1. Check both operands for negative values.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option -consider-analysis-perimeter-as-trust-boundary.

## Examples

### Modulo with User Input

```
#include <stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem =  128%userden; //Noncompliant
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using a user input. The input is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

#### Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
#include<stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem = 0;
    if (userden > 0 ) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_INT_MOD`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Integer division by zero` | `Tainted division operand` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Tainted NULL or non-null-terminated string

Argument is from an unsecure source and might be NULL or not NULL-terminated

## Description

This defect occurs when strings from unsecure sources are used in string manipulation routines that implicitly dereference the string buffer, for instance, `strcpy` or `sprintf`.

**Tainted NULL or non-null-terminated string** raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

### Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

### Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Getting String from Input

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));//Noncompliant
    print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

**Correction — Validate the Data**

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // Noncompliant-TAINTED_STRING only flagged here
        // - string is not null
        // - string has a positive and limited size
        // - TAINTED_STRING on strlen used as a firewall
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

**Correction — Validate the Data**

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_STRING`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Tainted string format` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Tainted sign change conversion

Value from an unsecure source changes sign

## Description

This defect occurs when values from unsecure sources are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
memmove
strncmp
strncpy
calloc
malloc
memalign
```

### Risk

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

### Fix

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size); //Noncompliant
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

#### Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

### Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_SIGN_CHANGE`
**Impact:** Medium

## Version History

**Introduced in R2015b**

## See Also

`Sign change integer conversion overflow` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Tainted string format

Input format argument is from an unsecure source

## Description

This defect occurs when `printf`-style functions use a format specifier constructed from unsecure sources.

### Risk

If you use externally controlled elements to format a string, you can cause buffer overflow or data-representation problems. An attacker can use these string formatting elements to view the contents of a stack using `%x` or write to a stack using `%n`.

### Fix

Pass a static string to format string functions. This fix ensures that an external actor cannot control the string.

Another possible fix is to allow only the expected number of arguments. If possible, use functions that do not support the vulnerable `%n` operator in format strings.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Get Elements from User Input

```
#include <stdio.h>
#include <unistd.h>
#define MAX 40
void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf(userstr);//Noncompliant
}
```

This example prints the input argument `userstr`. The string is unknown. If it contains elements such as `%`, `printf` can interpret `userstr` as a string format instead of a string, causing your program to crash.

#### Correction — Print as String

One possible correction is to print `userstr` explicitly as a string so that there is no ambiguity.

```
#include "stdio.h"
#include <unistd.h>
```

```
#define MAX 40

void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf("%.20s", userstr);
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_STRING_FORMAT`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Tainted NULL or non-null-terminated string` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Tainted size of variable length array

Size of the variable-length array (VLA) is from an unsecure source and might be zero, negative, or too large

## Description

This defect occurs when the size of a variable length array (VLA) is obtained from an unsecure source.

### Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

### Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Example — User Input Argument Used as Size of VLA

```
#include<stdio.h>
#inclule<stdlib.h>
#define LIM 40

long squaredSum(int size) {

    int tabvla[size];
    long res = 0;
    for (int i=0 ; i<LIM-1 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

**Correction — Check VLA Size**

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 0 and less than 40, before creating the VLA

```
#include <stdio.h>
#include <stdlib.h>
#define LIM 40

long squaredSum(int size) {
    long res = 0;
    if (size>0 && size<LIM){
        int tabvla[size];
        for (int i=0 ; i<size || i<LIM-1 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }else{
        res = -1;
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_VLA_SIZE`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Memory allocation with tainted size` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Use of externally controlled environment variable

Value of environment variable is from an unsecure source

## Description

This defect occurs when functions that add or change environment variables, such as `putenv` and `setenv`, obtain new environment variable values from unsecure sources.

### Risk

If the environment variable is tainted, an attacker can control your system settings. This control can disrupt an application or service in potentially malicious ways.

### Fix

Before using the new environment variable, check its value to avoid giving control to external users.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Set Path in Environment

```
#define _XOPEN_SOURCE
#define _GNU_SOURCE
#include "stdlib.h"

void taintedenvvariable(void)
{
    char* path = getenv("APP_PATH");
    putenv(path); //Noncompliant
}
```

In this example, `putenv` changes an environment variable. The path `path` has not been checked to make sure that it is the intended path.

#### Correction — Sanitize Path

One possible correction is to sanitize the path, checking that it matches what you expect.

```
#define _POSIX_C_SOURCE
#include <stdlib.h>
#include <string.h>

/* Function to sanitize a path */
const char * sanitize_path(const char* str) {
    /* secure allowlist of paths */
    static const char *const authorized_paths[] = {
        "/bin",
        "/usr/bin"
    };
    if (str != NULL) {
        for (int i = 0; i < sizeof(authorized_paths) / sizeof(authorized_paths[0]); i++)
        if (strcmp(authorized_paths[i], str) == 0) {
            return authorized_paths[i];
        }
    }
    return NULL;
}

void taintedenvvariable(void)
{
    const char* path = getenv("APP_PATH");
    path = sanitize_path(path);
    if (path != NULL) {
        if (setenv("PATH", path, /* overwrite = */1) != 0) {
            /* fatal error */
            exit(1);
        }
    }
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_ENV_VARIABLE`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Execution of externally controlled command` | `Host change using externally controlled elements` | `Command executed from externally controlled path` | `Library loaded from externally controlled path` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

# Use of tainted pointer

Pointer from an unsecure source may be NULL or point to unknown memory

## Description

This defect occurs when:

- Tainted NULL pointer — the pointer obtained from an unsecure source is not validated against NULL.
- Tainted size pointer — the size of the memory zone that an unsecure pointer points to is not validated.

**Note** On a single pointer, your code can have instances of **Use of tainted pointer**, **Pointer dereference with tainted offset**, and **Tainted NULL or non-null-terminated string**. Bug Finder raises only the first tainted pointer defect that it finds.

### Risk

An attacker can give your program a pointer that points to unexpected memory locations. If the pointer is dereferenced to write, the attacker can:

- Modify the state variables of a critical program.
- Cause your program to crash.
- Execute unwanted code.

If the pointer is dereferenced to read, the attacker can:

- Read sensitive data.
- Cause your program to crash.
- Modify a program variable to an unexpected value.

### Fix

Avoid use of pointers from external sources.

Alternatively, if you trust the external source, sanitize the pointer before dereference. In a separate sanitization function:

- Check that the pointer is not NULL.
- Check the size of the memory location (if possible). This second check validates whether the size of the data the pointer points to matches the size your program expects.

The defect still appears in the body of the sanitization function. However, if you use a sanitization function, instead of several occurrences, the defect appears only once. You can justify the defect and hide it in later reviews by using code annotations. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Function That Dereferences an External Pointer

```
#include<stdlib.h>
void taintedptr(void) {
    char *p = getenv("ARG");
    char x = *(p+10);//Noncompliant
}
```

In this example, the pointer `*p` points to an string of unknown size. During the dereferencing operation, the pointer might be null or point to unknown memory, which can result in segmentation fault.

#### Correction — Check Pointer

One possible correction is to sanitize the pointer before using it. This example checks whether the pointer is `nullptr` before it is dereferenced.

```
#include<stdlib.h>
#include <string.h>
void taintedptr(void) {
    char *p = getenv("ARG");
    if(p!=NULL && strlen(p)>10)
    {
    char x = *(p+10);
    }
}
```

## Result Information

**Group:** Tainted Data
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `TAINTED_PTR`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Pointer dereference with tainted offset` | `Find defects (-checkers)` | `-consider-analysis-perimeter-as-trust-boundary`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Sources of Tainting in a Polyspace Analysis"
"Modify Default Behavior of Bug Finder Checkers"

16

# Concurrency Defects

# Asynchronously cancellable thread

Calling thread might be cancelled in an unsafe state

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This defect occurs when you use `pthread_setcanceltype` with argument `PTHREAD_CANCEL_ASYNCHRONOUS` to set the cancellability type of a calling thread to asynchronous (or immediate) . An asynchronously cancellable thread can be cancelled at any time, usually immediately upon receiving a cancellation request.

### Risk

The calling thread might be cancelled in an unsafe state that could result in a resources leak, a deadlock, a data race, data corruption, or unpredictable behavior.

### Fix

Remove the call to `pthread_setcanceltype` with argument `PTHREAD_CANCEL_ASYNCHRONOUS` to use the default cancellability type `PTHREAD_CANCEL_DEFERRED` instead. With the default cancellability type, the thread defers cancellation requests until it calls a function that is a cancellation point.

## Examples

### Cancellability Type of Thread Set to Asynchronous

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static int fatal_error(void)
{
    exit(1);
}


volatile int a = 5;
volatile int b = 10;

pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;

void* swap_values_thread(void* dummy)
{
    int i;
    int c;
    int result;
    if ((result =
             pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &i)) != 0) {
        /* handle error */
        fatal_error();
    }
    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
        c = b;
        b = a;
        a = c;
        if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
    }
    return NULL;
}

int main(void)
{
    int result;
    pthread_t worker;

    if ((result = pthread_create(&worker, NULL, swap_values_thread, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }

    /* Additional code */

    if ((result = pthread_cancel(worker)) != 0) {
        /* handle error */
        fatal_error();
    }


    if ((result = pthread_join(worker, 0)) != 0) {
        /* handle error */
        fatal_error();
    }

    if ((result = pthread_mutex_lock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }
    printf("a: %i | b: %i", a, b);
    if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }

    return 0;
}
```

In this example, the cancellability type of the `worker` thread is set to asynchronous. The mutex `global_lock` helps ensure that the `worker` and `main` threads do not access variables `a` and `b` at the same time. However, the `worker` thread might be cancelled while holding `global_lock`, and the `main` thread will never acquire `global_lock`, which results in a deadlock.

#### Correction — Use the Default Cancellability Type

One possible correction is to remove the call to `pthread_setcanceltype`. By default, the cancellability type of a new thread is set to `PTHREAD_CANCEL_DEFERRED`. The `worker` thread defers cancellation requests until it calls a function that is a cancellation point.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static int fatal_error(void)
{
    exit(1);
}


volatile int a = 5;
volatile int b = 10;

pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;

void* swap_values_thread(void* dummy)
{
    int i;
    int c;
    int result;
    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
        c = b;
        b = a;
        a = c;
        if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
    }
    return NULL;
}

int main(void)
{
    int result;
    pthread_t worker;

    if ((result = pthread_create(&worker, NULL, swap_values_thread, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }

    /* Additional code */

    if ((result = pthread_cancel(worker)) != 0) {
        /* handle error */
        fatal_error();
    }


    if ((result = pthread_join(worker, 0)) != 0) {
        /* handle error */
        fatal_error();
    }

    if ((result = pthread_mutex_lock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }
    printf("a: %i | b: %i", a, b);
    if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }

    return 0;
}
```

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `ASYNCHRONOUSLY_CANCELLABLE_THREAD`
**Impact:** Medium

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"

“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

### External Websites

POS47-C

# Atomic load and store sequence not atomic

Variable accessible between load and store operations

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.

This defect occurs when you use these functions to load, and then store an atomic variable.

- C functions:
  - `atomic_load()`
  - `atomic_load_explicit()`
  - `atomic_store()`
  - `atomic_store_explicit()`
- C++ functions:
  - `std::atomic_load()`
  - `std::atomic_load_explicit()`
  - `std::atomic_store()`
  - `std::atomic_store_explicit()`
  - `std::atomic::load()`
  - `std::atomic::store()`

A thread cannot interrupt an atomic load or an atomic store operation on a variable, but a thread can interrupt a store, and then load sequence.

### Risk

A thread can modify a variable between the load and store operations, resulting in a data race condition.

### Fix

To read, modify, and store a variable atomically, use a compound assignment operator such as +=, `atomic_compare_exchange()` or `atomic_fetch_*`-family functions.

## Examples

### Loading Then Storing an Atomic Variable

```
#include <stdatomic.h>
#include <stdbool.h>
```

```
static atomic_bool flag = ATOMIC_VAR_INIT(false);

void init_flag(void)
{
    atomic_init(&flag, false);
}

void toggle_flag(void)
{
    bool temp_flag = atomic_load(&flag);
    temp_flag = !temp_flag;
    atomic_store(&flag, temp_flag);
}

bool get_flag(void)
{
    return atomic_load(&flag);
}
```

In this example, variable `flag` of type `atomic_bool` is referenced twice inside the `toggle_flag()` function. The function loads the variable, negates its value, then stores the new value back to the variable. If two threads call `toggle_flag()`, the second thread can access `flag` between the load and store operations of the first thread. `flag` can end up in an incorrect state.

#### Correction — Use Compound Assignment to Modify Variable

One possible correction is to use a compound assignment operator to toggle the value of `flag`. The C standard defines the operation by using ^= as atomic.

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void toggle_flag(void)
{
    flag ^= 1;
}

bool get_flag(void)
{
    return flag;
}
```

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `ATOMIC_VAR_SEQUENCE_NOT_ATOMIC`
**Impact:** Medium

# Version History

**Introduced in R2018b**

## See Also

`Atomic variable accessed twice in an expression` | `Data race` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Atomic variable accessed twice in an expression

Variable can be modified between accesses

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.

This defect occurs when C atomic types or C++ `std::atomic` class variables appear twice in an expression and there are:

- Two atomic read operations on the variable.
- An atomic read and a distinct atomic write operation on the variable.

The C standard defines certain operations on atomic variables that are thread safe and do not cause data race conditions. Unlike individual operations, a pair of operations on the same atomic variable in an expression is not thread safe.

### Risk

A thread can modify the atomic variable between the pair of atomic operations, which can result in a data race condition.

### Fix

Do not reference an atomic variable twice in the same expression.

## Examples

### Referencing Atomic Variable Twice in an Expression

```
#include <stdatomic.h>

atomic_int n = ATOMIC_VAR_INIT(0);

int compute_sum(void)
{
    return n * (n + 1) / 2;
}
```

In this example, the global variable `n` is referenced twice in the return statement of `compute_sum()`. The value of `n` can change between the two distinct read operations. `compute_sum()` can return an incorrect value.

#### Correction — Pass Variable as Function Argument

One possible correction is to pass the variable as a function argument `n`. The variable is copied to memory and the read operations on the copy guarantee that `compute_sum()` returns a correct result. If you pass a variable of type `int` instead of type `atomic_int`, the correction is still valid.

```
#include <stdatomic.h>

int compute_sum(atomic_int n)
{
    return n * (n + 1) / 2;
}
```

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `ATOMIC_VAR_ACCESS_TWICE`
**Impact:** Medium

# Version History

**Introduced in R2018b**

## See Also

`Atomic load and store sequence not atomic` | `Data race` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Automatic or thread local variable escaping from a thread

Variable is passed from one thread to another without ensuring that variable stays alive through duration of latter thread

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.

This defect occurs when an automatic or thread local variable is passed by address from one thread to another thread without ensuring that the variable stays alive through the duration of the latter thread.

The defect checker applies to both C11 and POSIX threads.

### Risk

An automatic or thread local variable is allocated on the stack at the beginning of a thread and its lifetime extends till the end of the thread. The variable is not guaranteed to be alive when a different thread accesses it.

For instance, consider the start function of a C11 thread with these lines:

```
int start_thread(thrd_t *tid) {
   int aVar = 0;
   if(thrd_success != thrd_create(tid, start_thread_child, &aVar) {
     ...
   }
}
```

The `thrd_create` function creates a child thread with start function `start_thread_child` and passes the address of the automatic variable `aVar` to this function. When this child thread accesses `aVar`, the parent thread might have completed execution and `aVar` is no longer on the stack. The access might result in reading unpredictable values.

### Fix

When you pass a variable from one thread to another, make sure that the variable lifetime matches or exceeds the lifetime of both threads. You can achieve this synchronization in one of these ways:

- Declare the variable `static` so that it does not go out of stack when the current thread completes execution.
- Dynamically allocate the storage for the variable so that it is allocated on the heap instead of the stack and must be explicitly deallocated. Make sure that the deallocation happens after both threads complete execution.

These solutions require you to create a variable in nonlocal memory. Instead, you can use other solutions such as the `shared` keyword with OpenMP's threading interface that allows you to safely share local variables across threads.

## Examples

### Automatic or Thread-Local Variable Escaping Thread

```
#include <threads.h>
#include <stdio.h>

int create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return 0;
}

void create_parent_thread(thrd_t *tid, int *parentPtr) {
   if (thrd_success != thrd_create(tid, create_child_thread, parentPtr)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  int parentVal = 1;

  create_parent_thread(&tid, &parentVal);


  if (thrd_success != thrd_join(tid, NULL)) {
    /* Handle error */
  }
  return 0;
}
```

In this example, the value `parentVal` is local to the parent thread that starts in `main` and continues into the function `create_parent_thread`. However, in the body of `create_parent_thread`, the address of this local variable is passed to a child thread (the thread with start routine `create_child_thread`). The parent thread might have completed execution and the variable `parentVal` might have gone out of scope when the child thread accesses this variable.

The same issue appears if the variable is declared as thread-local, for instance with the C11 keyword `_Thread_local` (or `thread_local`):

```
_Thread_local int parentVal = 1;
```

#### Correction – Use Static Variables

One possible correction is to declare the variable `parentVal` as `static` so that the variable is on the stack for the entire duration of the program.

```
#include <threads.h>
#include <stdio.h>

int create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return 0;
}

void create_parent_thread(thrd_t *tid, int *parentPtr) {
   if (thrd_success != thrd_create(tid, create_child_thread, parentPtr)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  static int parentVal = 1;

  create_parent_thread(&tid, &parentVal);


  if (thrd_success != thrd_join(tid, NULL)) {
    /* Handle error */
  }
  return 0;
}
```

**Correction – Use Dynamic Memory Allocation**

Another possible correction is to dynamically allocate storage for variables to be shared across threads and explicitly free the storage after the threads complete execution.

```
#include <threads.h>
#include <stdio.h>

int create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return 0;
}

void create_parent_thread(thrd_t *tid, int *parentPtr) {
   if (thrd_success != thrd_create(tid, create_child_thread, parentPtr)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  int parentPtr = (int*) malloc(sizeof(int));

  if(parentPtr) {
      create_parent_thread(&tid, parentPtr);


      if (thrd_success != thrd_join(tid, NULL)) {
        /* Handle error */
      }
      free(parentPtr);
  }
  return 0;
}
```

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `LOCAL_ADDR_ESCAPE_THREAD`
**Impact:** Medium

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Blocking operation while holding lock

Task performs lengthy operation while holding a lock

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.

This defect occurs when a task (thread) performs a potentially lengthy operation while holding a lock.

The checker considers calls to these functions as potentially lengthy:

- Functions that access a network such as `recv`
- System call functions such as `fork`, `pipe` and `system`
- Functions for I/O operations such as `getchar` and `scanf`
- File handling functions such as `fopen`, `remove` and `lstat`
- Directory manipulation functions such as `mkdir` and `rmdir`

The checker automatically detects certain primitives that hold and release a lock, for instance, `pthread_mutex_lock` and `pthread_mutex_unlock`. For the full list of primitives that are automatically detected, see "Auto-Detection of Thread Creation and Critical Section in Polyspace".

### Risk

If a thread performs a lengthy operation when holding a lock, other threads that use the lock have to wait for the lock to be available. As a result, system performance can slow down or deadlocks can occur.

### Fix

Perform the blocking operation before holding the lock or after releasing the lock.

Some functions detected by this checker can be called in a way that does not make them potentially lengthy. For instance, the function `recv` can be called with the parameter `O_NONBLOCK` which causes the call to fail if no message is available. When called with this parameter, `recv` does not wait for a message to become available.

## Examples

### Network I/O Operations with `recv` While Holding Lock

```
#include <pthread.h>
#include <sys/socket.h>

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void thread_foo(void *ptr) {
  unsigned int num;
  int result;
  int sock;

  /* sock is a connected TCP socket */

  if ((result = pthread_mutex_lock(&mutex)) != 0) {
    /* Handle Error */
  }

  if ((result = recv(sock, (void *)&num, sizeof(unsigned int), 0)) < 0) {
    /* Handle Error */
  }

  /* ... */

  if ((result = pthread_mutex_unlock(&mutex)) != 0) {
    /* Handle Error */
  }
}

int main() {
  pthread_t thread;
  int result;

  if ((result = pthread_mutexattr_settype(
      &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_init(&mutex, &attr)) != 0) {
    /* Handle Error */
  }

  if (pthread_create(&thread, NULL,(void*(*)(void*))& thread_foo, NULL) != 0) {
    /* Handle Error */
  }

  /* ... */

  pthread_join(thread, NULL);

  if ((result = pthread_mutex_destroy(&mutex)) != 0) {
    /* Handle Error */
  }

  return 0;
}
```

In this example, in each thread created with `pthread_create`, the function `thread_foo` performs a network I/O operation with `recv` after acquiring a lock with `pthread_mutex_lock`. Other threads using the same lock variable `mutex` have to wait for the operation to complete and the lock to become available.

#### Correction — Perform Blocking Operation Before Acquiring Lock

One possible correction is to call `recv` before acquiring the lock.

```
#include <pthread.h>
#include <sys/socket.h>

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void thread_foo(void *ptr) {
  unsigned int num;
  int result;
  int sock;

  /* sock is a connected TCP socket */
  if ((result = recv(sock, (void *)&num, sizeof(unsigned int), 0)) < 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_lock(&mutex)) != 0) {
    /* Handle Error */
  }

    /* ... */

  if ((result = pthread_mutex_unlock(&mutex)) != 0) {
    /* Handle Error */
  }
}

int main() {
  pthread_t thread;
  int result;

  if ((result = pthread_mutexattr_settype(
      &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_init(&mutex, &attr)) != 0) {
    /* Handle Error */
  }

  if (pthread_create(&thread, NULL,(void*(*)(void*))& thread_foo, NULL) != 0) {
    /* Handle Error */
  }

  /* ... */

  pthread_join(thread, NULL);

  if ((result = pthread_mutex_destroy(&mutex)) != 0) {
    /* Handle Error */
  }

  return 0;
}
```

## Result Information

**Group:** Concurrency

**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `BLOCKING_WHILE_LOCKED`
**Impact:** Low

## Version History

**Introduced in R2018b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Multiple threads waiting on same condition variable

Using `cnd_signal` to wake up one of the threads might result in indefinite blocking

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.

This defect occurs when you use `cnd_signal` family functions to wake up one of at least two threads that are concurrently waiting on the same condition variable. For threads with the same priority level, `cnd_signal` family functions cause the thread scheduler to arbitrarily wake up on of the threads waiting on the condition variable that you signal with the `cnd_signal` family function.

The checkers flags the `cnd_signal` family function call. See the **Event** column in the **Results Details** pane to view the threads waiting on the same condition variable.

### Risk

The thread that is woken up with a `cnd_signal` family function usually tests for a condition predicate. While the condition predicate is false, the thread waits again on the condition variable until it is woken up by another thread that signals the condition variable. It is possible that the program ends up in a state where no thread is available to signal the condition variable, which results in indefinite blocking.

### Fix

Use `cnd_broadcast` family functions instead to wake all threads waiting on the condition variable, or use a different condition variable for each thread.

## Examples

### Use of `cnd_signal` to Wake Up One of Many Threads Waiting on Condition Variable

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <threads.h>

typedef int thrd_return_t;

static void fatal_error(void)
{
    exit(1);
}

enum { NTHREADS = 5 };

mtx_t mutex;
cnd_t cond;
```

```
thrd_return_t next_step(void* t)
{
    static size_t current_step = 0;
    size_t my_step = *(size_t*)t;

    if (thrd_success != mtx_lock(&mutex)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu has the lock\n", my_step);
    while (current_step != my_step) {
        printf("Thread %zu is sleeping...\n", my_step);
        if (thrd_success !=
            cnd_wait(&cond, &mutex)) {
            /* Handle error */
            fatal_error();
        }
        printf("Thread %zu woke up\n", my_step);
    }
    /* Do processing ... */
    printf("Thread %zu is processing...\n", my_step);
    current_step++;

    /* Signal a waiting task */
    if (thrd_success !=
        cnd_signal(&cond)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu is exiting...\n", my_step);

    if (thrd_success != mtx_unlock(&mutex)) {
        /* Handle error */
        fatal_error();
    }
    return (thrd_return_t)0;
}

int main(void)
{
    thrd_t threads[NTHREADS];
    size_t step[NTHREADS];

    if (thrd_success != mtx_init(&mutex, mtx_plain)) {
        /* Handle error */
        fatal_error();
    }
    if (thrd_success != cnd_init(&cond)) {
        /* Handle error */
        fatal_error();
    }
    /* Create threads */
    for (size_t i = 0; i < NTHREADS; ++i) {
        step[i] = i;
        if (thrd_success != thrd_create(&threads[i],
                                        next_step,
                                        &step[i])) {
            /* Handle error */
            fatal_error();
        }
    }
    /* Wait for all threads to complete */
    for (size_t i = NTHREADS; i != 0; --i) {
        if (thrd_success != thrd_join(threads[i - 1], NULL)) {
            /* Handle error */
            fatal_error();
        }
    }
    (void)mtx_destroy(&mutex);
    (void)cnd_destroy(&cond);
    return 0;
}
```

In this example, multiple threads are created and assigned step level. Each thread checks if its assigned step level matches the current step level (condition predicate). If the predicate is false, the thread goes back to waiting on the condition variable `cond`. The use of `cnd_signal` to signal the `cond` causes the thread scheduler to arbitrarily wake up one of the threads waiting on `cond`. This can result in indefinite blocking when the condition predicate of woken up thread is false and no other thread is available to signal `cond`.

**Correction — Use `cnd_broadcast` to Wake up All the Threads**

One possible correction is to use `cnd_broadcast` instead to signal `cond`. The function `cnd_signal` wakes up all the thread that are waiting on `cond`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <threads.h>

typedef int thrd_return_t;

static void fatal_error(void)
{
    exit(1);
}

enum { NTHREADS = 5 };

mtx_t mutex;
cnd_t cond;

thrd_return_t next_step(void* t)
{
    static size_t current_step = 0;
    size_t my_step = *(size_t*)t;

    if (thrd_success != mtx_lock(&mutex)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu has the lock\n", my_step);
    while (current_step != my_step) {
        printf("Thread %zu is sleeping...\n", my_step);
        if (thrd_success !=
            cnd_wait(&cond, &mutex)) {
            /* Handle error */
            fatal_error();
        }
        printf("Thread %zu woke up\n", my_step);
    }
    /* Do processing ... */
    printf("Thread %zu is processing...\n", my_step);
    current_step++;

    /* Signal a waiting task */
    if (thrd_success !=
        cnd_broadcast(&cond)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu is exiting...\n", my_step);

    if (thrd_success != mtx_unlock(&mutex)) {
        /* Handle error */
        fatal_error();
    }
    return (thrd_return_t)0;
}

int main_test_next_step(void)
{
    thrd_t threads[NTHREADS];
    size_t step[NTHREADS];

    if (thrd_success != mtx_init(&mutex, mtx_plain)) {
        /* Handle error */
        fatal_error();
    }
    if (thrd_success != cnd_init(&cond)) {
        /* Handle error */
        fatal_error();
    }
    /* Create threads */
    for (size_t i = 0; i < NTHREADS; ++i) {
        step[i] = i;
        if (thrd_success != thrd_create(&threads[i],
                                        next_step,
                                        &step[i])) {
            /* Handle error */
            fatal_error();
        }
    }
    /* Wait for all threads to complete */
    for (size_t i = NTHREADS; i != 0; --i) {
        if (thrd_success != thrd_join(threads[i - 1], NULL)) {
            /* Handle error */
            fatal_error();
        }
    }
    (void)mtx_destroy(&mutex);
    (void)cnd_destroy(&cond);
    return 0;
}
```

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SIGNALED_COND_VAR_NOT_UNIQUE`
**Impact:** Low

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)` | `Function that can spuriously fail not wrapped in loop` | `Function that can spuriously wake up not wrapped in loop` | `Data race` | `Missing unlock` | `Missing lock` | `Deadlock` | `Multiple mutexes with one conditional variable`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

### External Websites

CON38-C

# Data race

Multiple tasks perform unprotected nonatomic operations on shared variable

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This defect occurs when:

1. Multiple tasks perform unprotected operations on a shared variable.
2. At least one task performs a write operation.
3. At least one operation is nonatomic. To detect data race on both atomic and nonatomic operations, use the options `-detect-atomic-data-race`. See "Extend Data Race Checkers to Atomic Operations".

   See "Define Atomic Operations in Multitasking Code".

The defect checker reports one data race result for each shared variable with conflicting read-write or write-write operations.

If you activate this checker without specifying the multitasking options first, you see a warning in the log:To find this defect, specify the multitasking options before analysis.

To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see "Configuring Polyspace Multitasking Analysis Manually".

### Risk

Data race can result in unpredictable values of the shared variable because you do not control the order of the operations in different tasks.

Data races between two write operations are more serious than data races between a write and read operation. Two write operations can interfere with each other and result in indeterminate values. To identify write-write conflicts, use the filters on the **Detail** column of the **Results List** pane. For these conflicts, the **Detail** column shows the additional line:

```
 Variable value may be altered by write-write concurrent access.
```

See also "Filter and Group Results in Polyspace Desktop User Interface" or "Filter and Sort Results in Polyspace Access Web Interface".

### Fix

To fix this defect, protect the operations on the shared variable using critical sections, temporal exclusion or another means. See "Protections for Shared Variables in Multitasking Code".

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon. For an example, see below.

### Extend Checker

Extend this checker to check for data races in operations that Bug Finder might not detect by default. For instance:

- You might be using multithreading functions that are not supported by Polyspace. Extend this checker by mapping the functions of your multithreading functions to their known POSIX equivalent. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.
- Polyspace assumes that certain operations are atomic and excludes them from data race checks. See “Define Atomic Operations in Multitasking Code”. These assumptions might not apply to your environment. To extend the data race checkers to include these operations, use the option `-detect-atomic-data-race`. See “Extend Data Race Checkers to Atomic Operations”.

Note that the checker reports one data race result per shared variable. The event list for the result shows upto fifteen conflicting read-write and write-write conflicts. You cannot customize these limits.

## Examples

### Unprotected Operation on Global Variable from Multiple Tasks

```
int var;
void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
    var++;
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Specification | |
|---|---|---|
| `Configure multitasking manually` | ☑ | |
| `Tasks (-entry-points)` | `task1`<br>`task2`<br>`task3` | |
| `Critical section details (-critical-section-begin -critical-section-end)` | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2,task3
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

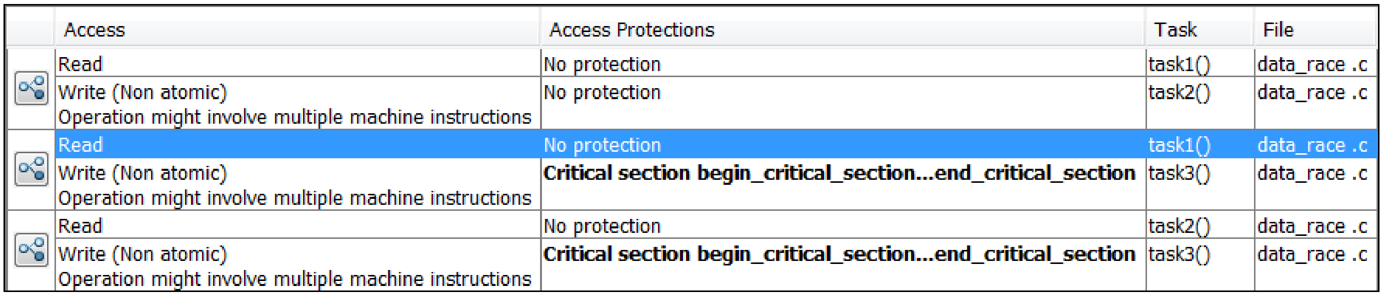
In this example, the tasks `task1`, `task2`, and `task3` call the function `increment`. `increment` contains the operation `var++` that can involve multiple machine instructions including:

- Reading `var`.
- Writing an increased value to `var`.

These machine instructions, when executed from `task1` and `task2`, can occur concurrently in an unpredictable sequence. For example, reading `var` from `task1` can occur either before or after writing to `var` from `task2`. Therefore the value of `var` can be unpredictable.

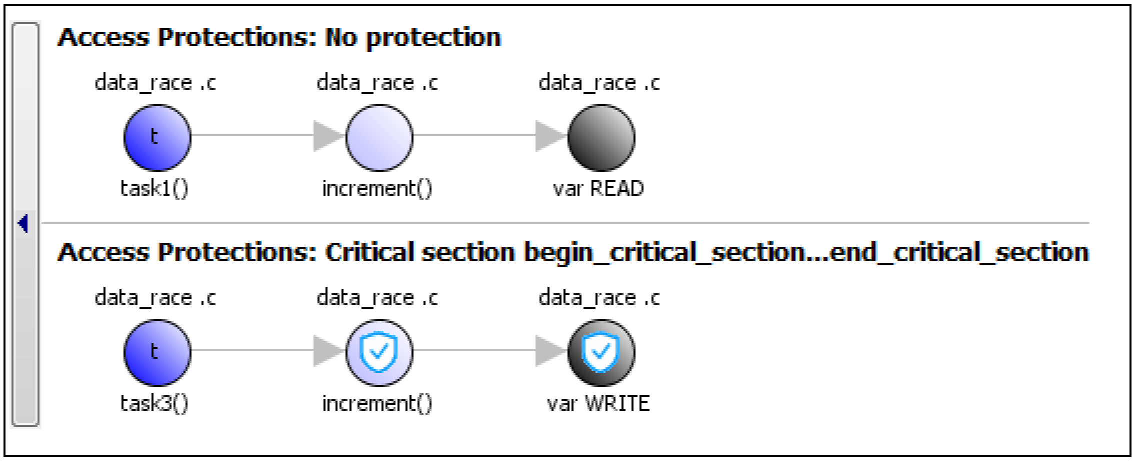
Though `task3` calls `increment` inside a critical section, other tasks do not use the same critical section. The operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

Therefore, the three tasks are operating on a shared variable without common protection. In your result details, you see each pair of conflicting function calls.



| | Access | Access Protections | Task | File |
|---|---|---|---|---|
| | Read | No protection | task1() | data_race .c |
| | Write (Non atomic)<br>Operation might involve multiple machine instructions | No protection | task2() | data_race .c |
| | Read | No protection | task1() | data_race .c |
| | Write (Non atomic)<br>Operation might involve multiple machine instructions | **Critical section begin_critical_section...end_critical_section** | task3() | data_race .c |
| | Read | No protection | task2() | data_race .c |
| | Write (Non atomic)<br>Operation might involve multiple machine instructions | **Critical section begin_critical_section...end_critical_section** | task3() | data_race .c |

If you click the icon, you see the function call sequence starting from the entry point to the read or write operation. You also see that the operation starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.



Access Protections: No protection

Access Protections: Critical section begin_critical_section...end_critical_section

**Correction — Place Operation in Critical Section**

One possible correction is to place the operation in critical section. You can implement the critical section in multiple ways. For instance:

- You can place `var++` in a critical section. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The operation `var++` from the three tasks cannot interfere with each other.

  To implement the critical section, in the function `increment`, place the operation `var++` between calls to `begin_critical_section` and `end_critical_section`.

  ```
  int var;

  void begin_critical_section(void);
  void end_critical_section(void);

  void increment(void) {
        begin_critical_section();
        var++;
        end_critical_section();
  }

  void task1(void)  {
        increment();
  }

  void task2(void)  {
        increment();
  }

  void task3(void)  {
        increment();
  }
  ```
- You can place the call to `increment` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `increment` from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call `increment` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      var++;
}

void task1(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task2(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

#### Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

| Option | Value |
|---|---|
| Temporally exclusive tasks (-temporal-exclusions-file) | task1 task2 task3 |

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

### Unprotected Operation in Threads Created with `pthread_create`

```
#include <pthread.h>
```

```
pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    count = count + 1;
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    c = count;
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

In this example, Bug Finder detects the creation of separate threads with `pthread_create`. The **Data race** defect is raised because the operation `count = count + 1` in the thread with id `thread_increment` conflicts with the operation `c = count` in the thread with id `thread_get`. The variable `count` is accessed in multiple threads without a common protection.

The two conflicting operations are nonatomic. The operation `c = count` is nonatomic on 32-bit targets. See “Define Atomic Operations in Multitasking Code”.

#### Correction — Protect Operations with `pthread_mutex_lock` and `pthread_mutex_unlock` Pair

To prevent concurrent access on the variable `count`, protect operations on `count` with a critical section. Use the functions `pthread_mutex_lock` and `pthread_mutex_unlock` to implement the critical section.

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}
```

```
void* set_count(void *args)
{
    long long c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

### Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `DATA_RACE`
**Impact:** High

## Version History

**Introduced in R2014b**

### See Also

`Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)` | `Temporally exclusive tasks (-temporal-exclusions-file)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Tasks (-entry-points)` | `Configure multitasking manually` | `Target processor type (-target)` | `Find defects (-checkers)` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double lock` | `Double unlock` | `Missing lock` | `Missing unlock`

#### Topics

"Analyze Multitasking Programs in Polyspace"
"Protections for Shared Variables in Multitasking Code"
"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Define Atomic Operations in Multitasking Code"
"Extend Concurrency Defect Checkers to Unsupported Multithreading Environments"

“Extend Data Race Checkers to Atomic Operations”

# Data race on adjacent bit fields

Multiple threads perform unprotected operations on adjacent bit fields of a shared data structure

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This defect occurs when:

1. Multiple tasks perform unprotected operations on bit fields that are part of the same structure.

   For instance, a task operates on field `errorFlag1` and another task on field `errorFlag2` in a variable of this type:

   ```
   struct errorFlags {
      unsigned int errorFlag1 : 1;
      unsigned int errorFlag2 : 1;
      ...
   }
   ```

   Suppose that the operations are not atomic with respect to each other. In other words, you have not implemented protection mechanisms to ensure that one operation is completed before another operation begins.
2. At least one of the unprotected operations is a write operation.

To find this defect, before analysis, you must specify the multitasking options. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see "Configuring Polyspace Multitasking Analysis Manually".

### Risk

Adjacent bit fields that are part of the same structure might be stored in one byte in the same memory location. Read or write operations on all variables including bit fields occur one byte or word at a time. To modify only specific bits in a byte, steps similar to these steps occur in sequence:

1. The byte is loaded into RAM.
2. A mask is created so that only specific bits are modified to the intended value and the remaining bits remain unchanged.
3. A bitwise OR operation is performed between the copy of the byte in RAM and the mask.
4. The byte with specific bits modified is copied back from RAM.

When you access two different bit fields, these four steps have to be performed for each bit field. If the accesses are not protected, all four steps for one bit field might not be completed before the four steps for the other bit field begin. As a result, the modification of one bit field might undo the modification of an adjacent bit field. For instance, in the preceding example, the modification of `errorFlag1` and `errorFlag2` can occur in the following sequence. Steps 1,2 and 5 relate to modification of `errorFlag1` and while steps 3,4 and 6 relate to that of `errorFlag2`.

1. The byte with both `errorFlag1` and `errorFlag2` unmodified is copied into RAM, for purposes of modifying `errorFlag1`.

2 A mask that modifies only `errorFlag1` is bitwise OR-ed with this copy.

3 The byte containing both `errorFlag1` and `errorFlag2` unmodified is copied into RAM a second time, for purposes of modifying `errorFlag2`.

4 A mask that modifies only `errorFlag2` is bitwise OR-ed with this second copy.

5 The version with `errorFlag1` modified is copied back. This version has `errorFlag2` unmodified.

6 The version with `errorFlag2` modified is copied back. This version has `errorFlag1` unmodified and overwrites the previous modification.

#### Fix

To fix this defect, protect the operations on bit fields that are part of the same structure by using critical sections, temporal exclusion, or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon.

## Examples

### Unprotected Operation on Global Variable from Multiple Tasks

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

In this example, `task1` and `task2` access different bit fields `IOFlag` and `SetupFlag`, which belong to the same structured variable `InterruptConfigbitsProc12`.

To emulate multitasking behavior, specify the options listed in this table.

| Option | Specification |
|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ |
| **Tasks** on page 2-121 | task1<br>task2 |

At the command-line, use:

```
polyspace-bug-finder
  -entry-points task1,task2
```

#### Correction – Use Critical Sections

One possible correction is to wrap the bit field access in a critical section. A critical section lies between a call to a lock function and an unlock function. In this correction, the critical section lies between the calls to functions `begin_critical_section` and `end_critical_section`.

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void begin_critical_section(void);
void end_critical_section(void);

void task1 (void) {
    begin_critical_section();
    InterruptConfigbitsProc12.IOFlag = 0;
    end_critical_section();
}

void task2 (void) {
    begin_critical_section();
    InterruptConfigbitsProc12.SetupFlag = 0;
    end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify options listed in this table.

| Option | Specification |
|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ |

| Option | Specification | |
|---|---|---|
| **Tasks** on page 2-121 | `task1`<br>`task2` | |
| **Critical section details** on page 2-133 | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

At the command-line, use:

```
polyspace-bug-finder
  -entry-points task1,task2
  -critical-section-begin begin_critical_section:cs1
  -critical-section-end end_critical_section:cs1
```

#### Correction – Avoid Bit Fields

If you do not have memory constraints, use the `char` data type instead of bit fields. The `char` variables in a structure occupy at least one byte and do not have the thread-safety issues that come from bit manipulations in a byte-sized operation. Data races do not result from unprotected operations on different `char` variables that are part of the same structure.

```
typedef struct
{
   unsigned char IOFlag;
   unsigned char InterruptFlag;
   unsigned char Register1Flag;
   unsigned char SignFlag;
   unsigned char SetupFlag;
   unsigned char Register2Flag;
   unsigned char ProcessorFlag;
   unsigned char GeneralFlag;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

Though the checker does not flag this correction, do not use this correction for C99 or earlier. Only from C11 and later does the C Standard mandate that distinct `char` variables cannot be accessed using the same word.

#### Correction – Insert Bit Field of Size 0

You can enter a non-bit field member or an unnamed bit field member of size 0 between two adjacent bit fields that might be accessed concurrently. A non-bit field member or size 0 bit field member ensures that the subsequent bit field starts from a new memory location. In this corrected example, the size 0 bit field member ensures that `IOFlag` and `SetupFlag` are stored in distinct memory locations.

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int : 0;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** DATA_RACE_BIT_FIELDS
**Impact:** High

# Version History

**Introduced in R2020b**

## See Also

Data race | Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts) | Temporally exclusive tasks (-temporal-exclusions-file) | Critical section details (-critical-section-begin -critical-section-end) | Tasks (-entry-points) | Configure multitasking manually | Target processor type (-target) | Find defects (-checkers) | Data race through standard library function call

### Topics

"Analyze Multitasking Programs in Polyspace"
"Protections for Shared Variables in Multitasking Code"
"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Define Atomic Operations in Multitasking Code"
"Extend Concurrency Defect Checkers to Unsupported Multithreading Environments"

# Data race through standard library function call

Multiple tasks make unprotected calls to thread-unsafe standard library function

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.

This defect occurs when:

1. Multiple tasks call the same standard library function.

   For instance, multiple tasks call the `strerror` function.
2. The calls are not protected using a common protection.

   For instance, the calls are not protected by the same critical section.

Functions flagged by this defect are not guaranteed to be reentrant. A function is reentrant if it can be interrupted and safely called again before its previous invocation completes execution. If a function is not reentrant, multiple tasks calling the function without protection can cause concurrency issues. For the list of functions that are flagged, see CON33-C: Avoid race conditions when using library functions.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see "Configuring Polyspace Multitasking Analysis Manually".

### Risk

The functions flagged by this defect are nonreentrant because their implementations can use global or static variables. When multiple tasks call the function without protection, the function call from one task can interfere with the call from another task. The two invocations of the function can concurrently access the global or static variables and cause unpredictable results.

The calls can also cause more serious security vulnerabilities, such as abnormal termination, denial-of-service attack, and data integrity violations.

### Fix

To fix this defect, do one of the following:

- Use a reentrant version of the standard library function if it exists.

  For instance, instead of `strerror()`, use `strerror_r()` or `strerror_s()`. For alternatives to functions flagged by this defect, see the documentation for CON33-C.
- Protect the function calls using common critical sections or temporal exclusion.

  See `Critical section details (-critical-section-begin -critical-section-end)` and `Temporally exclusive tasks (-temporal-exclusions-file)`.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon. For an example, see below.

## Examples

### Unprotected Call to Standard Library Function from Multiple Tasks

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Specification |
|---|---|
| `Configure multitasking manually` | ☑ |

| Option | Specification | |
|---|---|---|
| Tasks (-entry-points) | task1<br>task2<br>task3 | |
| Critical section details (-critical-section-begin -critical-section-end) | **Starting routine** | **Ending routine** |
| | begin_critical_section | end_critical_section |

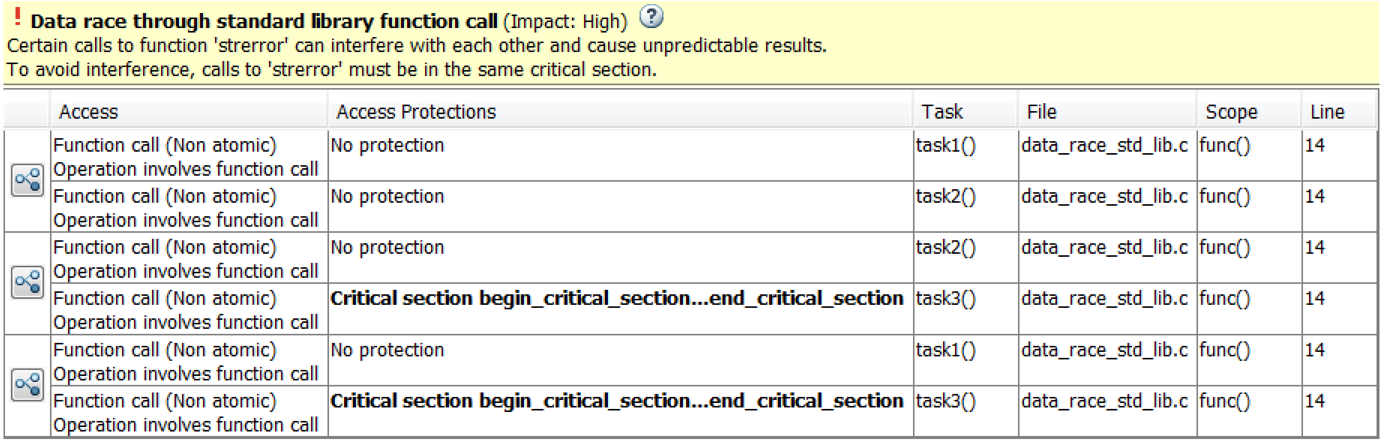
On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2,task3
  -critical-section-begin begin_critical_section:cs1
  -critical-section-end end_critical_section:cs1
```

In this example, the tasks, `task1`, `task2` and `task3`, call the function `func`. `func` calls the nonreentrant standard library function, `strerror`.

Though `task3` calls `func` inside a critical section, other tasks do not use the same critical section. Operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

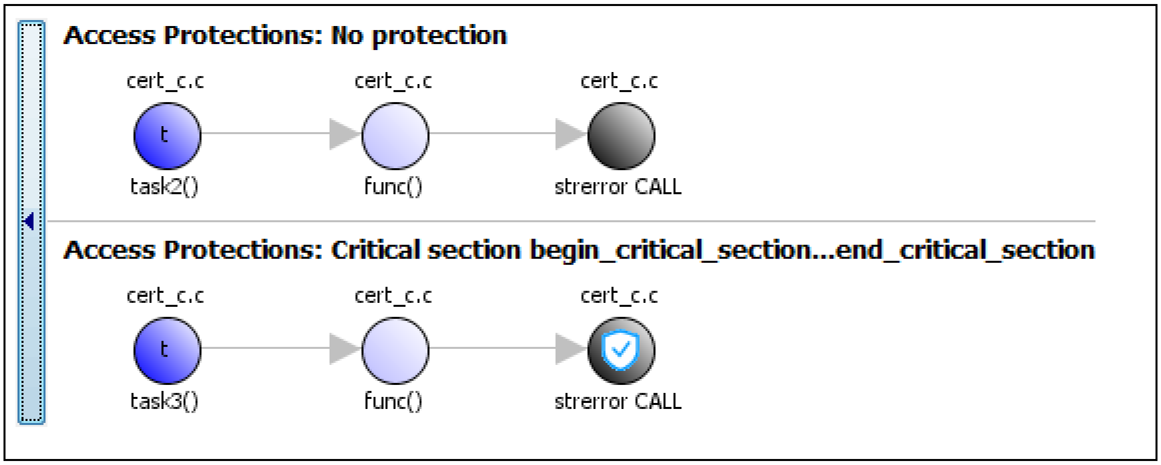
These three tasks are calling a nonreentrant standard library function without common protection. In your result details, you see each pair of conflicting function calls.



! **Data race through standard library function call** (Impact: High)
Certain calls to function 'strerror' can interfere with each other and cause unpredictable results.
To avoid interference, calls to 'strerror' must be in the same critical section.

| | Access | Access Protections | Task | File | Scope | Line |
|---|---|---|---|---|---|---|
| | Function call (Non atomic)<br>Operation involves function call | No protection | task1() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task2() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task2() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | **Critical section begin_critical_section...end_critical_section** | task3() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task1() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | **Critical section begin_critical_section...end_critical_section** | task3() | data_race_std_lib.c | func() | 14 |

If you click the icon, you see the function call sequence starting from the entry point to the standard library function call. You also see that the call starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.


Access Protections: No protection
cert_c.c
t
task2()
cert_c.c
func()
cert_c.c
strerror CALL
Access Protections: Critical section begin_critical_section...end_critical_section
cert_c.c
t
task3()
cert_c.c
func()
cert_c.c
strerror CALL

#### Correction — Use Reentrant Version of Standard Library Function

One possible correction is to use a reentrant version of the standard library function `strerror`. You can use the POSIX version `strerror_r` which has the same functionality but also guarantees thread-safety.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);
enum { BUFFERSIZE = 64 };

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char errmsg[BUFFERSIZE];
    if (strerror_r(errno, errmsg, BUFFERSIZE) != 0) {
      /* Handle error */
    }
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

**Correction — Place Function Call in Critical Section**

One possible correction is to place the call to `strerror` in critical section. You can implement the critical section in multiple ways.

For instance, you can place the call to the intermediate function `func` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `func` and therefore the calls to `strerror` from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call `func` between calls to `begin_critical_section` and `end_critical_section`.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    begin_critical_section();
    func(fptr1);
    end_critical_section();
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     begin_critical_section();
     func(fptr2);
     end_critical_section();
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

#### Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

| Option | Value |
|---|---|
| `Temporally exclusive tasks (-temporal-exclusions-file)` | `task1 task2 task3` |

On the command-line, you can use the following:

```
polyspace-bug-finder
    -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

## Result Information

**Group:** Concurrency<br>
**Language:** C | C++<br>
**Default:** On<br>
**Command-Line Syntax:** `DATA_RACE_STD_LIB`<br>
**Impact:** High

# Version History

**Introduced in R2016b**

## See Also

`Temporally exclusive tasks (-temporal-exclusions-file)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Tasks (-entry-points)` | `Configure multitasking manually` | `Find defects (-checkers)` | `Data race` | `Destruction of locked mutex` | `Double lock` | `Double unlock` | `Missing lock` | `Missing unlock`

### Topics

"Analyze Multitasking Programs in Polyspace"<br>
"Protections for Shared Variables in Multitasking Code"<br>
"Interpret Bug Finder Results in Polyspace Desktop User Interface"<br>
"Interpret Bug Finder Results in Polyspace Access Web Interface"<br>
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"<br>
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Deadlock

Call sequence to lock functions cause two tasks to block each other

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This defect occurs when multiple tasks are stuck in their critical sections (CS) because:

- Each CS waits for another CS to end.
- The critical sections (CS) form a closed cycle. For example:
  - CS #1 waits for CS #2 to end, and CS #2 waits for CS #1 to end.
  - CS #1 waits for CS #2 to end, CS #2 waits for CS #3 to end and CS #3 waits for CS #1 to end.

Polyspace expects critical sections of code to follow a specific format. A critical section lies between a call to a lock function and a call to an unlock function. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function. Both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

### Risk

Each task waits for a critical section in another task to end and is unable to proceed. The program can freeze indefinitely.

### Fix

The fix depends on the root cause of the defect. You can try to break the cyclic order between the tasks in one of these ways:

- Write down all critical sections involved in the deadlock in a certain sequence. Whenever you call the lock functions of the critical sections within a task, respect the order in that sequence. See an example below.
- If one of the critical sections involved in a deadlock occurs in an interrupt, try to disable all interrupts during critical sections in all tasks. See `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Reviewing this defect is an opportunity to check if all operations in your critical section are really meant to be executed as an atomic block. It is a good practice to keep critical sections at a bare minimum.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.

- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Extend Checker

You might be using multithreading functions that are not supported by Polyspace. Extend this checker by mapping the functions of your multithreading functions to their known POSIX equivalent. See "Extend Concurrency Defect Checkers to Unsupported Multithreading Environments".

## Examples

### Deadlock with Two Tasks

```
void task1(void);
void task2(void);

int var;
void perform_task_cycle(void) {
 var++;
}

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_2();
    begin_critical_section_1();
    perform_task_cycle();
    end_critical_section_1();
    end_critical_section_2();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| Configure multitasking manually | ☑ | |
| Tasks (-entry-points) | task1<br>task2 | |
| Critical section details (-critical-section-begin -critical-section-end) | **Starting routine** | **Ending routine** |
| | begin_critical_section_1 | end_critical_section_1 |
| | begin_critical_section_2 | end_critical_section_2 |

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `begin_critical_section_1`.
2. `task2` calls `begin_critical_section_2`.
3. `task1` reaches the instruction `begin_critical_section_2();`. Since `task2` has already called `begin_critical_section_2`, `task1` waits for `task2` to call `end_critical_section_2`.
4. `task2` reaches the instruction `begin_critical_section_1();`. Since `task1` has already called `begin_critical_section_1`, `task2` waits for `task1` to call `end_critical_section_1`.

#### Correction-Follow Same Locking Sequence in Both Tasks

One possible correction is to follow the same sequence of calls to lock and unlock functions in both `task1` and `task2`.

```
void task1(void);
void task2(void);
void perform_task_cycle(void);

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}
```

### Deadlock with More Than Two Tasks

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);


void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock3();
    lock1();
    performTaskCycle();
    unlock1();
    unlock3();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** | ☑ | |
| **Entry points** | `task1`<br>`task2`<br>`task3` | |
| **Critical section details** | **Starting routine** | **Ending routine** |
| | `lock1` | `unlock1` |
| | `lock2` | `unlock2` |
| | `lock3` | `unlock3` |

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `lock1`.
2. `task2` calls `lock2`.
3. `task3` calls `lock3`.
4. `task1` reaches the instruction `lock2();`. Since `task2` has already called `lock2`, `task1` waits for call to `unlock2`.
5. `task2` reaches the instruction `lock3();`. Since `task3` has already called `lock3`, `task2` waits for call to `unlock3`.
6. `task3` reaches the instruction `lock1();`. Since `task1` has already called `lock1`, `task3` waits for call to `unlock1`.

#### Correction — Break Cyclic Order

To break the cyclic order between critical sections, note every lock function in your code in a certain sequence, for example:

1. `lock1`
2. `lock2`
3. `lock3`

If you use more than one lock function in a task, use them in the order in which they appear in the sequence. For example, you can use `lock1` followed by `lock2` but not `lock2` followed by `lock1`.

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);

void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock1();
    lock3();
    performTaskCycle();
    unlock3();
    unlock1();
 }
}
```

### Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** DEADLOCK
**Impact:** High

## Version History

**Introduced in R2014b**

### See Also

Temporally exclusive tasks (-temporal-exclusions-file) | Critical section details (-critical-section-begin -critical-section-end) | Tasks (-entry-points) | Configure multitasking manually | Find defects (-checkers) | Data race | Data race through standard library function call | Destruction of locked mutex | Double lock | Double unlock | Missing lock | Missing unlock

**Topics**

"Analyze Multitasking Programs in Polyspace"
"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"

“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”
“Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”

# Destruction of locked mutex

Task tries to destroy a mutex in the locked state

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.

This defect occurs when a task destroys a mutex after it is locked (and before it is unlocked). The locking and destruction can happen in the same task or different tasks.

### Risk

A mutex is locked to protect shared variables from concurrent access. If a mutex is destroyed in the locked state, the protection does not apply.

### Fix

To fix this defect, destroy the mutex only after you unlock it. It is a good design practice to:

- Initialize a mutex *before* creating the threads where you use the mutex.
- Destroy a mutex *after* joining the threads that you created.

On the **Result Details** pane, you see two events, the locking and destruction of the mutex, and the tasks that initiated the events. To navigate to the corresponding line in your source code, click the event.

## Examples

### Locking and Destruction in Different Tasks

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
  pthread_mutex_unlock (&lock3);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_destroy (&lock3);
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

In this example, after task `t0` locks the mutex `lock3`, task `t1` can destroy it. The destruction occurs if the following events happen in sequence:

1. `t0` acquires `lock3`.
2. `t0` releases `lock2`.
3. `t0` releases `lock1`.
4. `t1` acquires the lock `lock1` released by `t0`.
5. `t1` acquires the lock `lock2` released by `t0`.
6. `t1` destroys `lock3`.

For simplicity, this example uses a mix of automatic and manual concurrency detection. The tasks `t0` and `t1` are manually specified as entry points by using the option `Tasks (-entry-points)`. The critical sections are implemented through primitives `pthread_mutex_lock` and `pthread_mutex_unlock` that the software detects automatically. In practice, for entry point specification (thread creation), you will use primitives such as `pthread_create`. The next example shows how the defect can appear when you use `pthread_create`.

**Correction — Place Lock-Unlock Pair Together in Same Critical Section as Destruction**

The locking and destruction of `lock3` occurs inside the critical section imposed by `lock1` and `lock2`, but the unlocking occurs outside. One possible correction is to place the lock-unlock pair in the same critical section as the destruction of the mutex. Use one of these critical sections:

- Critical section imposed by `lock1` alone.
- Critical section imposed by `lock1` and `lock2`.

In this corrected code, the lock-unlock pair and the destruction is placed in the critical section imposed by `lock1` and `lock2`. When `t0` acquires `lock1` and `lock2`, `t1` has to wait for their release before it executes the instruction `pthread_mutex_destroy (&lock3);`. Therefore, `t1` cannot destroy mutex `lock3` in the locked state.

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);

  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);

  pthread_mutex_destroy (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

#### Locking and Destruction in Start Routine of Thread

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_exit((void*) 0);
}

void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_destroy(&lock);
    pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Thread that initializes mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use mutex for atomic operation*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   pthread_exit(NULL);
}
```

In this example, four threads are created. The threads are assigned different actions.

- The first thread `callThd[0]` initializes the mutex `lock`.
- The second and third threads, `callThd[1]` and `callThd[2]`, perform an atomic operation protected by the mutex `lock`.
- The fourth thread `callThd[3]` destroys the mutex `lock`.

The threads can interrupt each other. Therefore, immediately after the second or third thread locks the mutex, the fourth thread can destroy it.

#### Correction — Initialize and Destroy Mutex Outside Start Routine

One possible correction is to initialize and destroy the mutex in the `main` function outside the start routine of the threads. The threads perform only the atomic operation. You need two fewer threads because the mutex initialization and destruction threads are not required.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 2
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_work(void *arg) {
   pthread_mutex_lock (&lock);
   atomic_operation();
   pthread_mutex_unlock (&lock);
   pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
   /* Initialize mutex */
   pthread_mutex_init(&lock, NULL);

   for(i=0; i<NUMTHREADS; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy mutex */
   pthread_mutex_destroy(&lock);

   pthread_exit(NULL);
}
```

#### Correction — Use A Second Mutex To Protect Lock-Unlock Pair and Destruction

Another possible correction is to use a second mutex and protect the lock-unlock pair from the destruction. This corrected code uses the mutex `lock2` to achieve this protection. The second mutex is initialized in the `main` function outside the start routine of the threads.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
pthread_mutex_t lock2;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}

void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_destroy(&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}
```

```
int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize second mutex */
   pthread_mutex_init(&lock2, NULL);

   /* Thread that initializes first mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use first mutex for atomic operation */
   /* The threads use second mutex to protect first from destruction in locked state*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys first mutex */
   /* The thread uses the second mutex to prevent destruction of locked mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);


   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy second mutex */
   pthread_mutex_destroy(&lock2);

   pthread_exit(NULL);
}
```

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** DESTROY_LOCKED
**Impact:** Medium

# Version History

**Introduced in R2016b**

## See Also

`Tasks (-entry-points)` | `Configure multitasking manually` | `Target processor type (-target)` | `Find defects (-checkers)` | `Data race` | `Data race through standard library function call` | `Deadlock` | `Double lock` | `Double unlock` | `Missing lock` | `Missing unlock`

### Topics

“Analyze Multitasking Programs in Polyspace”
“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Double lock

Lock function is called twice in a task without an intermediate call to unlock function

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

This defect occurs when:

- A task calls a lock function `my_lock`.
- The task calls `my_lock` again before calling the corresponding unlock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `task1` calls a lock function `lock`, other tasks calling `lock` must wait until `task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

### Risk

A call to a lock function begins a critical section so that other tasks have to wait to enter the same critical section. If the same lock function is called again within the critical section, the task blocks itself.

### Fix

The fix depends on the root cause of the defect. A double lock defect often indicates a coding error. Perhaps you omitted the call to an unlock function to end a previous critical section and started the next critical section. Perhaps you wanted to use a different lock function for the second critical section.

Identify each critical section of code, that is, the section that you want to be executed as an atomic block. Call a lock function at the beginning of the section. Within the critical section, make sure that you do not call the lock function again. At the end of the section, call the unlock function that corresponds to the lock function.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    ...
  }
  my_unlock();
}
```

```
void func2() {
  {
   my_lock();
   ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

You might be using locking functions that are not supported by Polyspace. Extend this checker by mapping your locking functions to its known POSIX equivalent. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.

## Examples

### Double Lock

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    lock();
    global_var += 1;
    lock();
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Value | |
|---|---|---|
| Configure multitasking manually | ☑ | |
| Tasks (-entry-points) | task1<br>task2 | |
| Critical section details (-critical-section-begin -critical-section-end) | **Starting routine** | **Ending routine** |
| | lock | unlock |

On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2
   -critical-section-begin lock:cs1
   -critical-section-end unlock:cs1
```

`task1` enters a critical section through the call `lock();`. `task1` calls `lock` again before it leaves the critical section through the call `unlock();`.

**Correction — Remove First Lock**

If you want the first `global_var+=1;` to be outside the critical section, one possible correction is to remove the first call to `lock`. However, if other tasks are using `global_var`, this code can produce a `Data race` error.

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    global_var += 1;
    lock();
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

**Correction — Remove Second Lock**

If you want the first `global_var+=1;` to be inside the critical section, one possible correction is to remove the second call to `lock`.

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    lock();
    global_var += 1;
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

**Correction — Add Another Unlock**

If you want the second `global_var+=1;` to be inside a critical section, another possible correction is to add another call to `unlock`.

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    lock();
    global_var += 1;
    unlock();
    lock();
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

### Double Lock with Function Call

```
int global_var;
```

```
void lock(void);
void unlock(void);

void performOperation(void) {
  lock();
  global_var++;
}

void task1(void)
{
    lock();
    global_var += 1;
    performOperation();
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| Configure multitasking manually | ☑ | |
| Tasks (-entry-points) | task1<br>task2 | |
| Critical section details (-critical-section-begin -critical-section-end) | **Starting routine** | **Ending routine** |
| | lock | unlock |

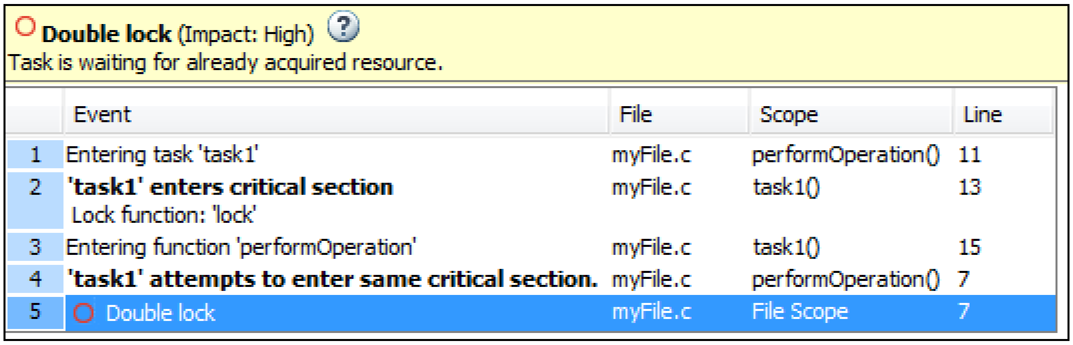
On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2
   -critical-section-begin lock:cs1
   -critical-section-end unlock:cs1
```

`task1` enters a critical section through the call `lock();`. `task1` calls the function `performOperation`. In `performOperation`, `lock` is called again even though `task1` has not left the critical section through the call `unlock();`.

In the result details for the defect, you see the sequence of instructions leading to the defect. For instance, you see that following the first entry into the critical section, the execution path:

- Enters function `performOperation`.
- Inside `performOperation`, attempts to enter the same critical section once again.



○ **Double lock** (Impact: High)

Task is waiting for already acquired resource.

| | Event | File | Scope | Line |
|---|---|---|---|---|
| 1 | Entering task 'task1' | myFile.c | performOperation() | 11 |
| 2 | **'task1' enters critical section**<br>Lock function: 'lock' | myFile.c | task1() | 13 |
| 3 | Entering function 'performOperation' | myFile.c | task1() | 15 |
| 4 | **'task1' attempts to enter same critical section.** | myFile.c | performOperation() | 7 |
| 5 | ○ Double lock | myFile.c | File Scope | 7 |

You can click each event to navigate to the corresponding line in the source code.

### Correction — Remove Second Lock

One possible correction is to remove the call to `lock` in `task1`.

```
int global_var;

void lock(void);
void unlock(void);

void performOperation(void) {
  global_var++;
}

void task1(void)
{
    lock();
    global_var += 1;
    performOperation();
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `DOUBLE_LOCK`
**Impact:** High

## Version History

**Introduced in R2014b**

## See Also

`Temporally exclusive tasks (-temporal-exclusions-file)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Tasks (-entry-points)` | `Configure multitasking manually` | `Find defects (-checkers)` | `Data race` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double unlock` | `Missing lock` | `Missing unlock`

### Topics

"Analyze Multitasking Programs in Polyspace"
"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Concurrency Defect Checkers to Unsupported Multithreading Environments"

# Double unlock

Unlock function is called twice in a task without an intermediate call to lock function

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This defect occurs when:

- A task calls a lock function `my_lock`.
- The task calls the corresponding unlock function `my_unlock`.
- The task calls `my_unlock` again. The task does not call `my_lock` a second time between the two calls to `my_unlock`.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `task1` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `task1` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

### Risk

A double unlock defect can indicate a coding error. Perhaps you wanted to call a different unlock function to end a different critical section. Perhaps you called the unlock function prematurely the first time and only the second call indicates the end of the critical section.

### Fix

The fix depends on the root cause of the defect.

Identify each critical section of code, that is, the section that you want to be executed as an atomic block. Call a lock function at the beginning of the section. Only at the end of the section, call the unlock function that corresponds to the lock function. Remove any other redundant call to the unlock function.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    ...
  }
  my_unlock();
}
```

```
void func2() {
  {
   my_lock();
   ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

You might be using unlocking functions that are not supported by Polyspace. Extend this checker by mapping your unlocking functions to its known POSIX equivalent. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.

## Examples

### Double Unlock

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Value | |
|---|---|---|
| Configure multitasking manually | ☑ | |
| Tasks (-entry-points) | task1<br>task2 | |
| Critical section details (-critical-section-begin -critical-section-end) | **Starting routine** | **Ending routine** |
| | BEGIN_CRITICAL_SECTION | END_CRITICAL_SECTION |

On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2
   -critical-section-begin BEGIN_CRITICAL_SECTION:cs1
   -critical-section-end END_CRITICAL_SECTION:cs1
```

`task1` enters a critical section through the call `BEGIN_CRITICAL_SECTION();`. `task1` leaves the critical section through the call `END_CRITICAL_SECTION();`. `task1` calls `END_CRITICAL_SECTION` again without an intermediate call to `BEGIN_CRITICAL_SECTION`.

#### Correction — Remove Second Unlock

If you want the second `global_var+=1;` to be outside the critical section, one possible correction is to remove the second call to `END_CRITICAL_SECTION`. However, if other tasks are using `global_var`, this code can produce a `Data race` error.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    global_var += 1;
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

#### Correction — Remove First Unlock

If you want the second `global_var+=1;` to be inside the critical section, one possible correction is to remove the first call to `END_CRITICAL_SECTION`.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    global_var += 1;
    END_CRITICAL_SECTION();
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

#### Correction — Add Another Lock

If you want the second `global_var+=1;` to be inside a critical section, another possible correction is to add another call to `BEGIN_CRITICAL_SECTION`.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `DOUBLE_UNLOCK`

**Impact:** High

## Version History

**Introduced in R2014b**

## See Also

`Temporally exclusive tasks (-temporal-exclusions-file)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Tasks (-entry-points)` | `Configure multitasking manually` | `Find defects (-checkers)` | `Data race` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double lock` | `Missing lock` | `Missing unlock`

### Topics

"Analyze Multitasking Programs in Polyspace"
"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Concurrency Defect Checkers to Unsupported Multithreading Environments"

# Function that can spuriously fail not wrapped in loop

Loop checks failure condition after possible spurious failure

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

This defect occurs when the following atomic compare and exchange functions that can fail spuriously are called from outside a loop.

- C atomic functions:
  - `atomic_compare_exchange_weak()`
  - `atomic_compare_exchange_weak_explicit()`
- C++ atomic functions:
  - `std::atomic<T>::compare_exchange_weak(T* expected, T desired)`
  - `std::atomic<T>::compare_exchange_weak_explicit(T* expected, T desired, std::memory_order succ, std::memory_order fail)`
  - `std::atomic_compare_exchange_weak(std::atomic<T>* obj, T* expected, T desired)`
  - `std::atomic_compare_exchange_weak_explicit(volatile std::atomic<T>* obj, T* expected, T desired, std::memory_order succ, std::memory_order fail)`

The functions compare the memory contents of the object representations pointed to by `obj` and `expected`. The comparison can spuriously return false even if the memory contents are equal. This spurious failure makes the functions faster on some platforms.

### Risk

An atomic compare and exchange function that spuriously fails can cause unexpected results and unexpected control flow.

### Fix

Wrap atomic compare and exchange functions that can spuriously fail in a loop. The loop checks the failure condition after a possible spurious failure.

## Examples

### `atomic_compare_exchange_weak()` Not Wrapped in Loop

```
#include <stdatomic.h>

extern void reset_count(void);
atomic_int count = ATOMIC_VAR_INIT(0);
```

```
void increment_count(void)
{
    int old_count = atomic_load(&count);
    int new_count;
    new_count = old_count + 1;
    if (!atomic_compare_exchange_weak(&count, &old_count, new_count))
        reset_count();

}
```

In this example, `increment_count()` uses `atomic_compare_exchange_weak()` to compare `count` and `old_count`. If the counts are equal, `count` is incremented to `new_count`. If they are not equal, the count is reset. When `atomic_compare_exchange_weak()` fails spuriously, the count is reset unnecessarily.

#### Correction — Wrap `atomic_compare_exchange_weak()` in a `while` Loop

One possible correction is to wrap the call to `atomic_compare_exchange_weak()` in a `while` loop. The loop checks the failure condition after a possible spurious failure.

```
#include <stdatomic.h>

extern void reset_count(void);
atomic_int count = ATOMIC_VAR_INIT(0);

void increment_count(void)
{
    int old_count = atomic_load(&count);
    int new_count;
    new_count = old_count + 1;

    do {
        reset_count();

    } while (!atomic_compare_exchange_weak(&count, &old_count, new_count));

}
```

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SPURIOUS_FAILURE_NOT_WRAPPED_IN_LOOP`
**Impact:** Low

# Version History

**Introduced in R2018b**

## See Also

`Function that can spuriously wake up not wrapped in loop` | `Returned value of a sensitive function not checked` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Function that can spuriously wake up not wrapped in loop

Loop checks wake-up condition after possible spurious wake-up

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.

This defect occurs when the following wait-on-condition functions are called from outside a loop:

- C functions:
  - `cnd_wait()`
  - `cnd_timedwait()`
- POSIX functions:
  - `pthread_cond_wait()`
  - `pthread_cond_timedwait()`
- C++ `std::condition_variable` and `std::condition_variable_any` class member functions:
  - `wait()`
  - `wait_until()`
  - `wait_for()`

Wait-on-condition functions pause the execution of the calling thread when a specified condition is met. The thread wakes up and resumes once another thread notifies it with `cnd_broadcast()` or an equivalent function. The wake-up notification can be spurious or malicious.

### Risk

If a thread receives a spurious wake-up notification and the condition of the wait-on-condition function is not checked, the thread can wake up prematurely. The wake-up can cause unexpected control flow, indefinite blocking of other threads, or denial of service.

### Fix

Wrap wait-on-condition functions that can wake up spuriously in a loop. The loop checks the wake-up condition after a possible spurious wake-up notification.

## Examples

### `cnd_wait()` Not Wrapped in Loop

```
#include <stdio.h>
```

```
#include <stddef.h>
#include <threads.h>

#define THRESHOLD 100

static mtx_t lock;
static cnd_t cond;

void func(int input)
{
    if (thrd_success != mtx_lock(&lock)) {
        /* Handle error */
    }
    /* test condition to pause thread */
    if (input > THRESHOLD) {
        if (thrd_success != cnd_wait(&cond, &lock)) {
            /* Handle error */
        }
    }
    /* Proceed if condition to pause does not hold */


    if (thrd_success != mtx_unlock(&lock)) {
        /* Handle error */
    }
}
```

In this example, the thread uses `cnd_wait()` to pause execution when `input` is greater than `THRESHOLD`. The paused thread can resume if another thread uses `cnd_broadcast()`, which notifies all the threads. This notification causes the thread to wake up even if the pause condition is still true.

#### Correction — Wrap `cnd_wait()` in a `while` Loop

One possible correction is to wrap `cnd_wait()` in a `while` loop. The loop checks the pause condition after the thread receives a possible spurious wake-up notification.

```
#include <stdio.h>
#include <stddef.h>
#include <threads.h>

#define THRESHOLD 100

static mtx_t lock;
static cnd_t cond;

void func(int input)
{
    if (thrd_success != mtx_lock(&lock)) {
        /* Handle error */
    }
    /* test condition to pause thread */
    while (input > THRESHOLD) {
        if (thrd_success != cnd_wait(&cond, &lock)) {
            /* Handle error */
        }
    }
    /* Proceed if condition to pause does not hold */


    if (thrd_success != mtx_unlock(&lock)) {
        /* Handle error */
    }
}
```

### Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SPURIOUS_WAKEUP_NOT_WRAPPED_IN_LOOP`
**Impact:** Low

## Version History

**Introduced in R2018b**

### See Also

`Function that can spuriously fail not wrapped in loop` | `Returned value of a sensitive function not checked` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Join or detach of a joined or detached thread

Thread that was previously joined or detached is joined or detached again

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

This defect occurs when:

- You try to join a thread that was previously joined or detached.
- You try to detach a thread that was previously joined or detached.

The **Result Details** pane describes if the thread was previously joined or detached and also shows previous related events.

For instance, a thread joined with `thrd_join` is then detached with `thrd_detach`:

```
thrd_t id;
...
thrd_join(id, NULL);
thrd_detach(id);
```

Note that a thread is considered as joined only if a previous thread joining is successful. For instance, the thread is not considered as joined in the `if` branch here:

```
thrd_t t;
...
if (thrd_success != thrd_join(t, 0)) {
    /* Thread not considered joined */
}
```

The analysis cannot detect cases where a joined thread detaches itself using, for instance, the `thrd_current()` function.

### Risk

The C11 standard (clauses 7.26.5.3 and 7.26.5.6) states that a thread shall not be joined or detached once it was previously joined or detached. Violating these clauses of the standard results in undefined behavior.

### Fix

Avoid joining a thread that was already joined or detached previously. Avoid detaching a thread that was already joined or detached.

## Examples

### Joining a Thread Followed by Detaching the Thread

```
#include <stddef.h>
```

```
#include <threads.h>
#include <stdlib.h>


extern int thread_func(void *arg);

int main (void)
{
  thrd_t t;

  if (thrd_success != thrd_create (&t, thread_func, NULL)) {
    /* Handle error */
    return 0;
  }

  if (thrd_success != thrd_join (t, 0)) {
    /* Handle error */
    return 0;
  }

  if (thrd_success != thrd_detach (t)) {
    /* Handle error */
    return 0;
  }

  return 0;
}
```

In this example, the use of `thrd_detach` on a thread that was previously joined with `thrd_join` leads to undefined behavior.

To avoid compilation errors when running Bug Finder on this example, specify the C11 standard with the option `C standard version (-c-version)`.

#### Correction – Avoid Detaching a Joined Thread

Remove the prior `thrd_join` or the subsequent `thrd_detach` statement. In this corrected version, the `thrd_detach` statement is removed.

```
#include <stddef.h>
#include <threads.h>
#include <stdlib.h>


extern int thread_func(void *arg);

int main (void)
{
  thrd_t t;

  if (thrd_success != thrd_create (&t, thread_func, NULL)) {
    /* Handle error */
    return 0;
  }

  if (thrd_success != thrd_join (t, 0)) {
    /* Handle error */
    return 0;
  }

  return 0;
}
```

### Joining Thread Created in Detached State

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success != pthread_attr_init(&attr)) {
        return 0;
    }

    if(thread_success != pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED)) {
        return 0;
    }

    if(thread_success != pthread_create(&id, &attr, thread_func, NULL)) {
            return 0;
    }

    if(thread_success != pthread_join(id, NULL)) {
            return 0;
    }

    return 0;
}
```

In this example, the thread attribute is assigned the state `PTHREAD_CREATE_DETACHED`. A thread created using this attribute is then joined.

#### Correction – Create Threads as Joinable

One possible correction is to create a thread with thread attribute assigned to the state `PTHREAD_CREATE_JOINABLE` and then join the thread.

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success != pthread_attr_init(&attr)) {
        return 0;
    }

    if(thread_success != pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE)) {
        return 0;
    }

    if(thread_success != pthread_create(&id, &attr, thread_func, NULL)) {
            return 0;
    }

    if(thread_success != pthread_join(id, NULL)) {
            return 0;
    }

    return 0;
}
```

### Result Information

**Group:** Concurrency
**Language:** C
**Default:** Off
**Command-Line Syntax:** `DOUBLE_JOIN_OR_DETACH`
**Impact:** Medium

## Version History

**Introduced in R2019b**

### See Also

`Use of undefined thread ID|Missing or double initialization of thread attribute`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing lock

Unlock function without lock function

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This defect occurs when a task calls an unlock function before calling the corresponding lock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait till `my_task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

### Risk

A call to an unlock function without a corresponding lock function can indicate a coding error. For instance, perhaps the unlock function does not correspond to the lock function that begins the critical section.

### Fix

The fix depends on the root cause of the defect. For instance, if the defect occurs because of a mismatch between lock and unlock function, check the lock-unlock function pair in your Polyspace analysis configuration and fix the mismatch.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

You might be using locking and unlocking functions that are not supported by Polyspace. Extend this checker by mapping these functions to their known POSIX equivalent. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.

## Examples

### Missing lock

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset(void)
{
  begin_critical_section();
  global_var = 0;
  end_critical_section();
}

void my_task(void)
{
  global_var += 1;
  end_critical_section();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| Configure multitasking manually | ☑ | |
| Tasks (-entry-points) | my_task, reset | |
| Critical section details (-critical-section-begin -critical-section-end) | **Starting routine** | **Ending routine** |
| | begin_critical_section | end_critical_section |

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, my_task and reset. my_task calls end_critical_section before calling begin_critical_section.

**Correction — Provide Lock**

One possible correction is to call the lock function begin_critical_section before the instructions in the critical section.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset(void)
{
  begin_critical_section();
  global_var = 0;
  end_critical_section();
}

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
    end_critical_section();
}
```

### Lock in Condition

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      if(index%10==0) {
        begin_critical_section();
        global_var ++;
      }
      end_critical_section();
      index++;
    }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| `Configure multitasking manually` | ☑ | |
| `Tasks (-entry-points)` | `my_task`, `reset` | |
| `Critical section details (-critical-section-begin -critical-section-end)` | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`.

In the `while` loop, `my_task` leaves a critical section through the call `end_critical_section();`. In an iteration of the `while` loop:

- If `my_task` enters the `if` condition branch, the critical section begins through a call to `begin_critical_section`.
- If `my_task` does not enter the `if` condition branch and leaves the `while` loop, the critical section does not begin. Therefore, a **Missing lock** defect occurs.
- If `my_task` does not enter the `if` condition branch and continues to the next iteration of the `while` loop, the unlock function `end_critical_section` is called again. A **Double unlock** defect occurs.

Because `numCycles` is a `volatile` variable, it can take any value. Any of the cases above are possible. Therefore, a **Missing lock** defect and a **Double unlock** defect appear on the call `end_critical_section`.

### Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `BAD_UNLOCK`
**Impact:** Medium

## Version History

**Introduced in R2014b**

### See Also

`Temporally exclusive tasks (-temporal-exclusions-file)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Tasks (-entry-points)` | `Configure multitasking manually` | `Find defects (-checkers)` | `Data race` | `Data` race through standard library function call|Deadlock|Destruction of locked mutex|Double lock|Double unlock|Missing unlock

**Topics**

"Configuring Polyspace Multitasking Analysis Manually"
"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Concurrency Defect Checkers to Unsupported Multithreading Environments"

# Missing or double initialization of thread attribute

Duplicated initialization of thread attributes or noninitialized thread attribute used in functions that expect initialized attributes

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This defect occurs during one of these situations:

- You initialize a thread attribute twice with a function such as `pthread_attr_init` without an intermediate call to a function such as `pthread_attr_destroy`.

  The function `pthread_attr_destroy` destroys a thread attribute object and enables the system to reclaim resources associated with the object.
- You use a noninitialized thread attribute in a function such as `pthread_create`, which expects an initialized attribute. A thread attribute might be noninitialized because it was never initialized previously or destroyed with the `pthread_attr_destroy` function.

  Noninitialized thread attributes are detected for all functions in the POSIX standard.

The **Result Details** pane describes whether the attribute is doubly initialized or noninitialized and also shows previous related events.

Note that a thread attribute is considered initialized only if the call to `pthread_attr_init` is successful. For instance, the thread attribute is not initialized in the `if` branch here:

```
pthread_attr_t attr;
int thread_success;

thread_success = pthread_attr_init(&attr);
if(thread_success != 0) {
   /* Thread attribute considered noninitialized */
}
```

The issue is also flagged if you do not check the return value from a call to `pthread_attr_init`.

### Risk

Initializing a thread attribute without destroying the previously initialized attribute or using noninitialized thread attributes leads to undefined behavior.

### Fix

Before using a thread attribute, initialize the attribute by using the `pthread_attr_init` function.

```
pthread_attr_t attr;
int thread_success;

/* Initialize attribute */
thread_success = pthread_attr_init(&attr);
```

```
if(thread_success != 0) {
   /* Handle initialization error */
}
...
/* Use attribute */
thread_sucess = pthread_create(&thr, &attr, &thread_start, NULL);
```

After initialization, destroy a thread attribute by using `pthread_attr_destroy` before initializing again:

```
pthread_attr_t attr;
int thread_success;

/* Destroy attribute */
thread_success = pthread_attr_destroy(&attr);
if(thread_success != 0) {
   /* Handle destruction error */
}
...
/* Reinitialize attribute */
thread_success = pthread_attr_init(&attr);
```

## Examples

### Use of Noninitialized Thread Attribute

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success == pthread_create(&id, &attr, thread_func, NULL)) {
    }

    return 0;
}
```

In this example, the attribute `attr` is not initialized before its use in the `pthread_create` call.

#### Correction – Initialize Thread Attribute Before Use

Before using a thread attribute in the `pthread_create` function, initialize the attribute with the `pthread_attr_init` function.

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);

int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success != pthread_attr_init(&attr)) {
        return 0;
    }

    if(thread_success == pthread_create(&id, &attr, thread_func, NULL)) {
    }

    return 0;
}
```

### Return Value from Thread Attribute Initialization Not Checked

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);

int main() {
    pthread_t id;
    pthread_attr_t attr;

    pthread_attr_init(&attr);

    if(thread_success == pthread_create(&id, &attr, thread_func, NULL)) {
    }

    return 0;
}
```

In this example, the return value of `pthread_attr_init` is not checked. If the thread attribute initialization fails, the error does not get handled. A possibly undefined thread attribute is later used in the `pthread_create` function.

#### Correction – Handle Errors from Thread Attribute Initialization

One possible correction is to use the thread attribute only if the return value from `pthread_attr_init` indicates successful initialization.

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);

int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success != pthread_attr_init(&attr)) {
        return 0;
    }

    if(thread_success == pthread_create(&id, &attr, thread_func, NULL)) {
    }
    return 0;
}
```

### Result Information

**Group:** Concurrency
**Language:** C
**Default:** Off
**Command-Line Syntax:** `BAD_THREAD_ATTRIBUTE`
**Impact:** Medium

## Version History

**Introduced in R2019b**

### See Also

`Use of undefined thread ID` | `Join or detach of a joined or detached thread`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing unlock

Lock function without unlock function

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.

This defect occurs when:

- A task calls a lock function.
- The task ends without a call to an unlock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task, `my_task`, calls a lock function, `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, before analysis, you must specify the multitasking options. On the **Configuration** pane, select **Multitasking**.

### Risk

An unlock function ends a critical section so that other waiting tasks can enter the critical section. A missing unlock function can result in tasks blocked for an unnecessary length of time.

### Fix

Identify the critical section of code, that is, the section that you want to be executed as an atomic block. At the end of this section, call the unlock function that corresponds to the lock function used at the beginning of the section.

There can be other reasons and corresponding fixes for the defect. Perhaps you called the incorrect unlock function. Check the lock-unlock function pair in your Polyspace analysis configuration and fix the mismatch.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
```

```
    ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Extend Checker

You might be using locking and unlocking functions that are not supported by Polyspace. Extend this checker by mapping these functions to their known POSIX equivalent. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.

## Examples

### Missing Unlock

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset()
{
    begin_critical_section();
    global_var = 0;
    end_critical_section();
}

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Value | |
|---|---|---|
| Configure multitasking manually | ☑ | |
| Tasks (-entry-points) | my_task, reset | |
| Critical section details (-critical-section-begin -critical-section-end) | **Starting routine** | **Ending routine** |
| | begin_critical_section | end_critical_section |

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`. `my_task` enters a critical section through the call `begin_critical_section();`. `my_task` ends without calling `end_critical_section`.

#### Correction — Provide Unlock

One possible correction is to call the unlock function `end_critical_section` after the instructions in the critical section.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset(void)
{
    begin_critical_section();
    global_var = 0;
    end_critical_section();
}

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
    end_critical_section();
}
```

### Unlock in Condition

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;
    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        global_var = 0;
        end_critical_section();
      }
      index++;
    }
}
```

In this example, to emulate multitasking behavior, specify the following options.

| Option | Specification | |
|---|---|---|
| Configure multitasking manually | ☑ | |
| Tasks (-entry-points) | my_task, reset | |
| Critical section details (-critical-section-begin -critical-section-end) | **Starting routine** | **Ending routine** |
| | begin_critical_section | end_critical_section |

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`.

In the `while` loop, `my_task` enters a critical section through the call `begin_critical_section();`. In an iteration of the `while` loop:

- If `my_task` enters the `if` condition branch, the critical section ends through a call to `end_critical_section`.
- If `my_task` does not enter the `if` condition branch and leaves the `while` loop, the critical section does not end. Therefore, a **Missing unlock** defect occurs.
- If `my_task` does not enter the `if` condition branch and continues to the next iteration of the `while` loop, the lock function `begin_critical_section` is called again. A **Double lock** defect occurs.

Because `numCycles` is a `volatile` variable, it can take any value. Any of the cases above is possible. Therefore, a **Missing unlock** defect and a **Double lock** defect appear on the call `begin_critical_section`.

#### Correction — Place Unlock Outside Condition

One possible correction is to call the unlock function `end_critical_section` outside the `if` condition.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        global_var=0;
      }
      end_critical_section();
      index++;
    }
}
```

#### Correction — Place Unlock in Every Conditional Branch

Another possible correction is to call the unlock function `end_critical_section` in every branches of the `if` condition.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        global_var=0;
        end_critical_section();
      }
      else
        end_critical_section();
      index++;
    }
}
```

### Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** On
**Command-Line Syntax:** `BAD_LOCK`
**Impact:** High

## Version History

**Introduced in R2014b**

### See Also

`Temporally exclusive tasks (-temporal-exclusions-file)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Tasks (-entry-points)` | `Configure multitasking manually` | `Find defects (-checkers)` | `Data race` | `Data race through standard library function call` | `Deadlock` | `Destruction of locked mutex` | `Double lock` | `Double unlock` | `Missing lock`

#### Topics

"Configuring Polyspace Multitasking Analysis Manually"
"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Extend Concurrency Defect Checkers to Unsupported Multithreading Environments"

# Multiple mutexes used with same condition variable

Threads using different mutexes when concurrently waiting on the same condition variable is undefined behavior

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This defect occurs when multiple threads use more than one mutex to concurrently wait on the same condition variable. A thread waits on a condition variable by calling the functions `pthread_cond_timedwait` or `pthread_cond_wait`. These functions take a condition variable and a locked mutex as arguments, and the condition variable is bound to that mutex when the thread waits on the condition variable.

The checkers flags the use of `pthread_cond_timedwait` or `pthread_cond_wait` in one of the threads. See the **Event** column in the **Results Details** pane to view the threads waiting on the same condition variable and using a different mutex.

### Risk

When a thread waits on a condition variable using a mutex, the condition variable is bound to that mutex. Any other thread using a different mutex to wait on the same condition variable is undefined behavior according to the POSIX standard.

### Fix

Use the same mutex argument for `pthread_cond_timedwait` or `pthread_cond_wait` when threads are concurrently waiting on the same condition variable, or use separate condition variables for each mutex.

## Examples

### Concurrent Waiting on Condition Variable with Multiple Mutexes

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define Thrd_return_t                    void *
#define __USE_XOPEN2K8



#define COUNT_LIMIT 5

static void fatal_error(void)
{
    exit(1);
}


pthread_mutex_t mutex1;
pthread_mutex_t mutex2;
pthread_mutex_t mutex3;
pthread_cond_t cv;

int count1 = 0, count2 = 0, count3 = 0;
#define DELAY 8

Thrd_return_t waiter1(void* arg)
{
    int ret;
    while (count1 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex1)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                 pthread_cond_wait(&cv, &mutex1)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count1 = %d\n", ++count1);
        if ((ret = pthread_mutex_unlock(&mutex1)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter2(void* arg)
{
    int ret;
    while (count2 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex2)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                 pthread_cond_wait(&cv, &mutex2)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count2 = %d\n", ++count2);
        if ((ret = pthread_mutex_unlock(&mutex2)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t signaler(void* arg)
{
    int ret;
    while ((count1 < COUNT_LIMIT) || (count2 < COUNT_LIMIT)) {
        sleep(1);
        printf("signaling\n");
        if ((ret = pthread_cond_broadcast(&cv)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter3(void* arg)
{
    int ret;
    while (count3 % COUNT_LIMIT != 0) {
        if ((ret = pthread_mutex_lock(&mutex3)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                 pthread_cond_wait(&cv, &mutex3)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count3 = %d\n", ++count3);
        if ((ret = pthread_mutex_unlock(&mutex3)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

int main(void)
{
    int ret;
    pthread_t thread1, thread2, thread3;

    pthread_mutexattr_t attr;

    if ((ret = pthread_mutexattr_init(&attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
        /* Handle error */
        fatal_error();
    }

    if ((ret = pthread_mutex_init(&mutex1, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex2, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex3, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_cond_init(&cv, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread1, NULL, &waiter1, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread2, NULL, &waiter2, NULL))) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread3, NULL, &signaler, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread1, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread2, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread3, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }

    while (1) { ; }

    return 0;
}
```

In this example, a different mutex is used to protect each `count` variable. Since all three `waiter` functions wait on the same condition variable `cv` with different mutexes, the call to `pthread_cond_wait` will succeed for one of the threads and the call will be undefined for the other two.

The checker raises a defect for function `waiter3` even though the function is not invoked directly or indirectly by a thread, entry-point, or interrupt. The analysis considers function `waiter3` called by the main program through its function address or an unidentified thread whose creation is the missing source code.

#### Correction — Use the Same Mutex for All Threads Waiting on Same Condition Variable

One possible correction is to pass the same mutex argument to all the call to `pthread_cond_wait` that are used to wait on the same condition variable.

```
 #include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define Thrd_return_t                    void *
#define __USE_XOPEN2K8



#define COUNT_LIMIT 5

static void fatal_error(void)
{
    exit(1);
}


pthread_mutex_t mutex;

pthread_cond_t cv;

int count1 = 0, count2 = 0, count3 = 0;
#define DELAY 8

Thrd_return_t waiter1(void* arg)
{
    int ret;
    while (count1 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                 pthread_cond_wait(&cv, &mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count1 = %d\n", ++count1);
        if ((ret = pthread_mutex_unlock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter2(void* arg)
{
    int ret;
    while (count2 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                 pthread_cond_wait(&cv, &mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count2 = %d\n", ++count2);
        if ((ret = pthread_mutex_unlock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t signaler(void* arg)
```

```
{
    int ret;
    while ((count1 < COUNT_LIMIT) || (count2 < COUNT_LIMIT)) {
        sleep(1);
        printf("signaling\n");
        if ((ret = pthread_cond_broadcast(&cv)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter3(void* arg)
{
    int ret;
    while (count3 % COUNT_LIMIT != 0) {
        if ((ret = pthread_mutex_lock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                 pthread_cond_wait(&cv, &mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count3 = %d\n", ++count3);
        if ((ret = pthread_mutex_unlock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}
/*
void user_task(void)
{
    (void)waiter3(NULL);
} */

int main(void)
{
    int ret;
    pthread_t thread1, thread2, thread3;

    pthread_mutexattr_t attr;

    if ((ret = pthread_mutexattr_init(&attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
        /* Handle error */
        fatal_error();
    }

    if ((ret = pthread_mutex_init(&mutex, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_cond_init(&cv, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread1, NULL, &waiter1, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread2, NULL, &waiter2, NULL))) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread3, NULL, &signaler, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread1, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread2, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread3, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }

    while (1) { ; }

    return 0;
}
```

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `MULTI_MUTEX_WITH_ONE_COND_VAR`
**Impact:** Medium

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)` | `Function that can spuriously fail not wrapped in loop` | `Function that can spuriously wake up not wrapped in loop` | `Data race` | `Missing unlock` | `Missing lock` | `Deadlock` | `Multiple threads waiting on same condition variable`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

### External Websites

POS53-C

# Signal call in multithreaded program

Program with multiple threads uses `signal` function

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

This defect occurs when you use the `signal()` function in a program with multiple threads.

### Risk

According to the C11 standard (Section 7.14.1.1), use of the `signal()` function in a multithreaded program is undefined behavior.

### Fix

Depending on your intent, use other ways to perform an asynchronous action on a specific thread.

## Examples

### Use of `signal()` Function to Terminate Loop in Thread

```
#include <signal.h>
#include <stddef.h>
#include <threads.h>

volatile sig_atomic_t flag = 0;

void handler(int signum) {
  flag = 1;
}

/* Runs until user sends SIGUSR1 */
int func(void *data) {
  while (!flag) {
    /* ... */
  }
  return 0;
}

int main(void) {
  signal(SIGINT, handler); /* Undefined behavior */
  thrd_t tid;

  if (thrd_success != thrd_create(&tid, func, NULL)) {
    /* Handle error */
  }
  /* ... */
  return 0;
}
```

In this example, the `signal` function is used to terminate a `while` loop in the thread created with `thrd_create`.

**Correction — Use `atomic_bool` Variable to Terminate Loop**

One possible correction is to use an `atomic_bool` variable that multiple threads can access. In the corrected example, the child thread evaluates this variable before every loop iteration. After completing the program, you can modify this variable so that the child thread exits the loop.

```
#include <stdatomic.h>
#include <stdbool.h>
#include <stddef.h>
#include <threads.h>

atomic_bool flag = ATOMIC_VAR_INIT(false);

int func(void *data) {
  while (!flag) {
    /* ... */
  }
  return 0;
}

int main(void) {
  thrd_t tid;

  if (thrd_success != thrd_create(&tid, func, NULL)) {
    /* Handle error */
  }
  /* ... */
  /* Set flag when done */
  flag = true;

  return 0;
}
```

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SIGNAL_USE_IN_MULTITHREADED_PROGRAM`
**Impact:** Low

# Version History

**Introduced in R2018b**

## See Also

`Function called from signal handler not asynchronous-safe` | `Signal call from within signal handler` | `MISRA C:2012 Rule 21.5` | `Find defects (-checkers)`

**Topics**

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Thread-specific memory leak

Dynamically allocated thread-specific memory not freed before end of thread

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This defect occurs when you do not free thread-specific dynamically allocated memory before the end of a thread.

To create thread-specific storage, you generally do these steps:

1. You create a key for thread-specific storage.
2. You create the threads.
3. In each thread, you allocate storage dynamically and then associate the key with this storage.

   After the association, you can read the stored data later using the key.
4. Before the end of the thread, you free the thread-specific memory using the key.

The checker flags execution paths in the thread where the last step is missing.

The checker works on these families of functions:

- `tss_get` and `tss_set` (C11)
- `pthread_getspecific` and `pthread_setspecific` (POSIX)

### Risk

The data stored in the memory is available to other processes even after the threads end (memory leak). Besides security vulnerabilities, memory leaks can shrink the amount of available memory and reduce performance.

### Fix

Free dynamically allocated memory before the end of a thread.

You can explicitly free dynamically allocated memory with functions such as `free`.

Alternatively, when you create a key, you can associate a destructor function with the key. The destructor function is called with the key value as argument at the end of a thread. In the body of the destructor function, you can free any memory associated with the key. If you use this method, Bug Finder still flags a defect. Ignore this defect with appropriate comments. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.

- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Examples

### Memory Not Freed at End of Thread

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;

  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */
  }
  print_data();
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }

  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

In this example, the start function of each thread `func` calls two functions:

- `add_data`: This function allocates storage dynamically and associates the storage with a key using the `tss_set` function.
- `print_data`: This function reads the stored data using the `tss_get` function.

At the points where `func` returns, the dynamically allocated storage has not been freed.

**Correction — Free Dynamically Allocated Memory Explicitly**

One possible correction is to free dynamically allocated memory explicitly before leaving the start function of a thread. See the highlighted change in the corrected version.

In this corrected version, a defect still appears on the `return` statement in the error handling section of `func`. The defect cannot occur in practice because the error handling section is entered only if dynamic memory allocation fails. Ignore this remaining defect with appropriate comments. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;

  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */
  }
  print_data();
  free(tss_get(key));
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }

  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

### Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** THREAD_MEM_LEAK
**Impact:** Medium

## Version History

**Introduced in R2018b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Use of signal to kill thread

Uncaught signal kills entire process instead of specific thread

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

This defect occurs when you use an uncaught signal to kill a thread. For instance, you use the POSIX function `pthread_kill` and send the signal `SIGTERM` to kill a thread.

### Risk

Sending a signal kills the entire process instead of just the thread that you intend to kill.

For instance, the `pthread_kill` specifications state that if the disposition of a signal is to terminate, this action affects the entire process.

### Fix

Use other mechanisms that are intended to kill specific threads.

For instance, use the POSIX function `pthread_cancel` to terminate a specific thread.

## Examples

### Use of `pthread_kill` to Terminate Threads

```
#include <signal.h>
#include <pthread.h>

void* func(void *foo) {
  /* Execution of thread */
}

int main(void) {
  int result;
  pthread_t thread;

  if ((result = pthread_create(&thread, NULL, func, 0)) != 0) {
  }
  if ((result = pthread_kill(thread, SIGTERM)) != 0) {
  }

  /* This point is not reached because the process terminates in pthread_kill() */

  return 0;
}
```

In this example, the `pthread_kill` function sends the signal `SIGTERM` to kill a thread. The signal kills the entire process instead of the thread previously created with `pthread_create`.

#### Correction — Use `pthread_cancel` to Terminate Threads

One possible correction is to use the `pthread_cancel` function. The `pthread_cancel` terminates a thread specified by its first argument at a specific cancellation point or immediately, depending on the thread's cancellation type.

```
#include <signal.h>
#include <pthread.h>

void* func(void *foo) {
  /* Execution of thread */
}

int main(void) {
  int result;
  pthread_t thread;

  if ((result = pthread_create(&thread, NULL, func, 0)) != 0) {
    /* Handle Error */
  }
  if ((result = pthread_cancel(thread)) != 0) {
    /* Handle Error */
  }

  /* Continue executing */

  return 0;
}
```

See also:

- `pthread_cancel` for more information on cancellation types.
- Pthreads for functions that are allowed to be cancellation points.

## Result Information

**Group:** Concurrency
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `THREAD_KILLED_WITH_SIGNAL`
**Impact:** Low

# Version History

**Introduced in R2018b**

## See Also

`Signal call in multithreaded program` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Use of undefined thread ID

Thread ID from failed thread creation used in subsequent thread functions

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.

This defect occurs when a thread creation function such as `pthread_create` fails but you continue to use the ID from the thread creation.

For instance, `pthread_join` uses an undefined thread ID after the previous thread creation failed. The nonzero return value from `pthread_create` indicates the failed thread creation.

```
pthread_t id;
if(0! = pthread_create(&id, attr, start_func, NULL)) {
    ...
    phread_join(id, NULL);
    ...
}
```

The issue is also flagged if you do not check the return value from a call to `pthread_create`.

### Risk

According to the POSIX standard, if thread creation fails, the contents of the thread ID are undefined. The use of an undefined thread ID can lead to unpredictable results.

The issue often indicates a programming error. For instance, it is possible that you tested for nonzero values to determine successful thread creation:

```
if(0 != pthread_create(&id, attr, start_func, NULL))
```

instead of zero:

```
if(0 == pthread_create(&id, attr, start_func, NULL))
```

### Fix

If the use of an undefined thread ID comes from a programming error, fix the error. Otherwise, remove the thread functions that are using the undefined ID.

## Examples

### Threads Joined After Failed Thread Creation

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);
```

```
int main() {
    pthread_t id;
    if(thread_success != pthread_create(&id, NULL, thread_func, NULL)) {
         if(thread_success == pthread_join(id, NULL)) {
         }
    }

    return 0;
}
```

In this example, if `pthread_create` returns a nonzero value, thread creation has failed. The value of `*id` is undefined. The subsequent call to `pthread_join` uses this undefined value.

**Correction – Join Threads After Successful Thread Creation**

One possible correction is to call `pthread_join` with the thread ID as argument only if `pthread_create` returns zero.

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    if(thread_success == pthread_create(&id, NULL, thread_func, NULL)) {
         if(thread_success == pthread_join(id, NULL)) {
         }
    }

    return 0;
}
```

### Return Value from Thread Creation Not Checked

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    pthread_create(&id, NULL, thread_func, NULL);
    if(thread_success == pthread_join(id, NULL)) {
    }

    return 0;
}
```

In this example, the return value of `pthread_create` is not checked. If thread creation fails, the error does not get handled. A possibly undefined thread ID is later used in the `pthread_join` function.

#### Correction – Handle Errors from Thread Creation

One possible correction is to use the ID from thread creation only if the return value from `pthread_create` indicates successful thread creation.

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    if(thread_success == pthread_create(&id, NULL, thread_func, NULL)) {
         if(thread_success == pthread_join(id, NULL)) {
         }
    }

    return 0;
}
```

## Result Information

**Group:** Concurrency
**Language:** C
**Default:** Off
**Command-Line Syntax:** `UNDEFINED_THREAD_ID`
**Impact:** Medium

# Version History

**Introduced in R2019b**

## See Also

`Missing or double initialization of thread attribute` | `Join or detach of a joined or detached thread`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

17

# Object Oriented Defects

# *this not returned in copy assignment operator

`operator=` method does not return a pointer to the current object

## Description

This defect occurs when assignment operators such as `operator=` and `operator+=` do not return a reference to `*this`, where `this` is a pointer to the current object. If the `operator=` method does not return `*this`, it means that `a=b` or `a.operator=(b)` is not returning the assignee `a` following the assignment.

For instance:

- The operator returns its parameter instead of a reference to the current object.

  That is, the operator has a form `MyClass & operator=(const MyClass & rhs) { ... return rhs; }` instead of `MyClass & operator=(const MyClass & rhs) { ... return *this; }`.
- The operator returns by value and not reference.

  That is, the operator has a form `MyClass operator=(const MyClass & rhs) { ... return *this; }` instead of `MyClass & operator=(const MyClass & rhs) { ... return *this; }`.

### Risk

Users typically expect object assignments to behave like assignments between built-in types and expect an assignment to return the assignee. For instance, a right-associative chained assignment `a=b=c` requires that `b=c` return the assignee `b` following the assignment. If your assignment operator behaves differently, users of your class can face unexpected consequences.

The unexpected consequences occur when the assignment is part of another statement. For instance:

- If the `operator=` returns its parameter instead of a reference to the current object, the assignment `a=b` returns `b` instead of `a`. If the `operator=` performs a partial assignment of data members, following an assignment `a=b`, the data members of `a` and `b` are different. If you or another user of your class read the data members of the return value and expect the data members of `a`, you might have unexpected results. For an example, see “Return Value of operator= Same as Argument” on page 17-3.
- If the `operator=` method returns `*this` by value and not reference, a copy of `*this` is returned. If you expect to modify the result of the assignment using a statement such as `(a=b).modifyValue()`, you modify a copy of `a` instead of `a` itself.

### Fix

Return `*this` from your assignment operators.

## Examples

### Return Value of operator= Same as Argument

```
class MyClass {
    public:
        MyClass(bool b, int i): m_b(b), m_i(i) {}
        const MyClass& operator=(const MyClass& obj) {
            if (&obj!=this) {
                /* Note: Only m_i is copied. m_b retains its original value. */
                m_i = obj.m_i;
            }
            return obj;
        }
        bool isOk() const { return m_b;}
        int getI() const { return m_i;}
    private:
        bool m_b;
        int m_i;
};

void main() {
        MyClass r0(true, 0), r1(false, 1);
        /* Object calling isOk is r0 and the if block executes. */
        if ( (r1 = r0).isOk()) {
            /* Do something */
        }
}
```

In this example, the operator operator= returns its current argument instead of a reference to *this.

Therefore, in main, the assignment r1 = r0 returns r0 and not r1. Because the operator= does not copy the data member m_b, the value of r0.m_b and r1.m_b are different. The following unexpected behavior occurs.

| What You Might Expect | What Actually Happens |
|---|---|
| • The statement (r1 = r0).isOk() returns r1.m_b which has value false<br>• The if block does not execute. | • The statement (r1 = r0).isOk() returns r0.m_b which has value true<br>• The if block executes. |

#### Correction — Return *this

One possible correction is to return *this from operator=.

```
class MyClass {
    public:
        MyClass(bool b, int i): m_b(b), m_i(i) {}
        const MyClass& operator=(const MyClass& obj) {
            if (&obj!=this) {
                /* Note: Only m_i is copied. m_b retains its original value. */
                m_i = obj.m_i;
            }
            return *this;
        }
        bool isOk() const { return m_b;}
        int getI() const { return m_i;}
    private:
        bool m_b;
        int m_i;
};

void main() {
        MyClass r0(true, 0), r1(false, 1);
        /* Object calling isOk is r0 and the if block executes. */
        if ( (r1 = r0).isOk()) {
            /* Do something */
        }
}
```

## Result Information

**Group:** Object oriented
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `RETURN_NOT_REF_TO_THIS`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Base class assignment operator not called

Copy assignment operator does not call copy assignment operators of base subobjects

## Description

This defect occurs when a derived class copy assignment operator does not call the copy assignment operator of its base class.

### Risk

If this defect occurs, unless you are initializing the base class data members explicitly in the derived class assignment operator, the operator initializes the members implicitly by using the default constructor of the base class. Therefore, it is possible that the base class data members do not get assigned the right values.

If users of your class expect your assignment operator to perform a complete assignment between two objects, they can face unintended consequences.

### Fix

Call the base class copy assignment operator from the derived class copy assignment operator.

Even if the base class data members are not `private`, and you explicitly initialize the base class data members in the derived class copy assignment operator, replace this explicit initialization with a call to the base class copy assignment operator. Otherwise, determine why you retain the explicit initialization.

## Examples

### Base Class Copy Assignment Operator Not Called

```
class Base0 {
public:
    Base0();
    virtual ~Base0();
    Base0& operator=(const Base0&);
private:
    int _i;
};

class Base1 {
public:
    Base1();
    virtual ~Base1();
    Base1& operator=(const Base1&);
private:
    int _i;
};

class Derived: public Base0, Base1 {
public:
    Derived();
    ~Derived();
    Derived& operator=(const Derived& d) {
        if (&d == this) return *this;
        Base0::operator=(d);
        _j = d._j;
        return *this;
    }
private:
    int _j;
};
```

In this example, the class `Derived` is derived from two classes `Base0` and `Base1`. In the copy assignment operator of `Derived`, only the copy assignment operator of `Base0` is called. The copy assignment operator of `Base1` is not called.

The defect appears on the copy assignment operator of the derived class. Following are some tips for navigating in the source code:

- To find the derived class definition, right-click the derived class name and select **Go To Definition**.
- To find the base class definition, first navigate to the derived class definition. In the derived class definition, right-click the base class name and select **Go To Definition**.
- To find the definition of the base class copy assignment operator, first navigate to the base class definition. In the base class definition, right-click the operator name and select **Go To Definition**.

#### Correction — Call Base Class Copy Assignment Operator

If you want your copy assignment operator to perform a complete assignment, one possible correction is to call the copy assignment operator of class `Base1`.

```
class Base0 {
public:
    Base0();
    virtual ~Base0();
    Base0& operator=(const Base0&);
private:
    int _i;
};

class Base1 {
public:
    Base1();
    virtual ~Base1();
    Base1& operator=(const Base1&);
private:
    int _i;
};

class Derived: public Base0, Base1 {
public:
    Derived();
    ~Derived();
    Derived& operator=(const Derived& d) {
        if (&d == this) return *this;
        Base0::operator=(d);
        Base1::operator=(d);
        _j = d._j;
        return *this;
    }
private:
    int _j;
};
```

## Result Information

**Group:** Object oriented
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `MISSING_BASE_ASSIGN_OP_CALL`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)` | `Copy constructor not called in initialization list`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Base class destructor not virtual

Class cannot behave polymorphically for deletion of derived class objects

## Description

This defect occurs when a class has virtual functions but not a virtual destructor.

### Risk

The presence of virtual functions indicates that the class is intended for use as a base class. However, if the class does not have a virtual destructor, it cannot behave polymorphically for deletion of derived class objects.

If a pointer to this class refers to a derived class object, and you use the pointer to delete the object, only the base class destructor is called. Additional resources allocated in the derived class are not released and can cause a resource leak.

### Fix

One possible fix is to always use a virtual destructor in a class that contains virtual functions.

## Examples

### Base Class Destructor Not Virtual

```
class Base {
        public:
                Base(): _b(0) {};
                virtual void update() {_b += 1;};
        private:
                int _b;
};

class Derived: public Base {
        public:
                Derived(): _d(0) {};
                ~Derived() {_d = 0;};
                virtual void update() {_d += 1;};
        private:
                int _d;
};
```

In this example, the class Base does not have a virtual destructor. Therefore, if a Base* pointer points to a Derived object that is allocated memory dynamically, and the delete operation is performed on that Base* pointer, the Base destructor is called. The memory allocated for the additional member _d is not released.

The defect appears on the base class definition. Following are some tips for navigating in the source code:

- To find classes derived from the base class, right-click the base class name and select **Search For All References**. Browse through each search result to find derived class definitions.
- To find if you are using a pointer or reference to a base class to point to a derived class object, right-click the base class name and select **Search For All References**. Browse through search results that start with `Base*` or `Base&` to locate pointers or references to the base class. You can then see if you are using a pointer or reference to point to a derived class object.

#### Correction — Make Base Class Destructor Virtual

One possible correction is to declare a `virtual` destructor for the class `Base`.

```
class Base {
        public:
                Base(): _b(0) {};
                virtual ~Base() {_b = 0;};
                virtual void update() {_b += 1;};
        private:
                int _b;
};

class Derived: public Base {
        public:
                Derived(): _d(0) {};
                ~Derived() {_d = 0;};
                virtual void update() {_d += 1;};
        private:
                int _d;
};
```

### Result Information

**Group:** Object oriented
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `DTOR_NOT_VIRTUAL`
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Find defects (-checkers)`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

#### External Websites

CERT C++ OOP52-CPP

# Bytewise operations on nontrivial class object

Value representations may be improperly initialized or compared

## Description

This defect occurs when you use C Standard library functions to perform bytewise operation on non-trivial or non-standard layout class type objects. For definitions of trivial and standard layout classes, see the C++ Standard (ISO/IEC 14882:2017), [class], paragraphs 6 and 7 respectively.

The checker raises a defect when:

- You initialize or copy non-trivial class type objects using these functions:

  - `std::memset`
  - `std::memcpy`
  - `std::strcpy`
  - `std::memmove`

  To check whether a class type is trivial, use the type-traits library function `std::is_trivial`, for instance:

  ```
  #include <iostream>
  #include <type_traits>

  class trivialClass {};

  void checkTrivial(){
      static_assert(std::is_trivial<trivialClass>::value,
                    "Class is not trivial");
  }
  ```

  It is not sufficient to check that the object type is trivially copyable (`std::is_trivially_copyable<>:value`). A trivially copyable object does not guarantee the class invariants hold when you use the object later in your program.
- You compare non-standard layout class type objects using these functions:

  - `std::memcmp`
  - `std::strcmp`

Note that an incomplete class can be potentially nontrivial.

The checker does not raise a defect if the bytewise operation is performed through an alias. For example no defect is raised in the bytewise comparison and copy operations in this code. The bytewise operations use `dptr` and `sptr`, the aliases of non-trivial or non-standard layout class objects `d` and `s`.

```
void func(NonTrivialNonStdLayout *d, const NonTrivialNonStdLayout *s)
{
   void* dptr = (void*)d;
   const void* sptr = (void*)s;
   // ...
   // ...
   // ...
   if (!std::memcmp(dptr, sptr, sizeof(NonTrivialNonStdLayout))) {
     (void)std::memcpy(dptr, sptr, sizeof(NonTrivialNonStdLayout));
      // ...
   }
}
```

### Risk

Performing bytewise comparison operations by using C Standard library functions on non-trivial or non-standard layout class type object might result in unexpected values due to implementation details. The object representation depends on the implementation details, such as the order of private and public members, or the use of virtual function pointer tables to represent the object.

Performing bytewise setting operations by using C Standard library functions on non-trivial or non-standard layout class type object can change the implementation details. The operation might result in abnormal program behavior or a code execution vulnerability. For instance, if the address of a member function is overwritten, the call to this function invokes an unexpected function.

### Fix

To perform bytewise operations non-trivial or non-standard layout class type object, use these C++ special member functions instead of C Standard library functions.

| C Standard Library Functions | C++ Member Functions |
|---|---|
| `std::memset` | Class constructor |
| `std::memcpy`<br>`std::strcpy`<br>`std::memmove` | Class copy constructor<br>Class move constructor<br>Copy assignment operator<br>Move assignment operator |
| `std::memcmp`<br>`std::strcmp` | `operator<()`<br>`operator>()`<br>`operator==()`<br>`operator!=()` |

## Examples

### Using `memset` with non-trivial class object

```
#include <cstring>
#include <iostream>
#include <utility>

class nonTrivialClass
{
    int scalingFactor;
    int otherData;
public:
    nonTrivialClass() : scalingFactor(1) {}
    void set_other_data(int i);
    int f(int i)
    {
        return i / scalingFactor;
    }
    // ...
};

void func()
{
    nonTrivialClass c;
    // ... Code that mutates c ...
    std::memset(&c, 0, sizeof(nonTrivialClass));
    std::cout << c.f(100) << std::endl;
}
```

In this example, `func()` uses `std::memset` to reinitialize non-trivial class object `c` after it is first initialized with its default constructor. This bytewise operation might not properly initialize the value representation of `c`.

**Correction — Define Function Template That Uses `std::swap`**

One possible correction is to define a function template `clear()` that uses `std::swap` to perform a swap operation. The call to `clear()`properly reinitializes object `c` by swapping the contents of `c` and default initialized object `empty`.

```
 #include <cstring>
#include <iostream>
#include <utility>

class nonTrivialClass
{
    int scalingFactor;
    int otherData;
public:
    nonTrivialClass() : scalingFactor(1) {}
    void set_other_data(int i);
    int f(int i)
    {
        return i / scalingFactor;
    }
    // ...
};

template <typename T>
T& clear(T& o)
{
    using std::swap;
    T empty;
    swap(o, empty);
    return o;
}

void func()
{
    nonTrivialClass c;
    // ... Code that mutates c ...

    clear(c);
    std::cout << c.f(100) << std::endl;
}
```

## Result Information

**Group:** Object Oriented
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `MEMOP_ON_NONTRIVIAL_OBJ`
**Impact:** Medium

## Version History

**Introduced in R2019b**

## See Also

`Find defects (-checkers)` | `Memory comparison of padding data` | `Memory comparison of strings` | `Copy of overlapping memory`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Conversion or deletion of incomplete class pointer

You delete or cast to a pointer to an incomplete class

## Description

This defect occurs when you delete or cast to a pointer to an incomplete class. An incomplete class is one whose definition is not visible at the point where the class is used.

For instance, the definition of class `Body` is not visible when the `delete` operator is called on a pointer to `Body`:

```
class Handle {
  class Body *impl;
public:
  ~Handle() { delete impl; }
  // ...
};
```

### Risk

When you delete a pointer to an incomplete class, it is not possible to call any nontrivial destructor that the class might have. If the destructor performs cleanup activities such as memory deallocation, these activities do not happen.

A similar problem happens, for instance, when you downcast to a pointer to an incomplete class (downcasting is casting from a pointer to a base class to a pointer to a derived class). At the point of downcasting, the relationship between the base and derived class is not known. In particular, if the derived class inherits from multiple classes, at the point of downcasting, this information is not available. The downcasting cannot make the necessary adjustments for multiple inheritance and the resulting pointer cannot be dereferenced.

A similar statement can be made for upcasting (casting from a pointer to derived class to a pointer to a base class).

### Fix

When you delete or downcast to a pointer to a class, make sure that the class definition is visible.

Alternatively, you can perform one of these actions:

- Instead of a regular pointer, use the `std::shared_ptr` type to point to the incomplete class.
- When downcasting, make sure that the result is valid. Write error-handling code for invalid results.

## Examples

### Deletion of Pointer to Incomplete Class

```
class Handle {
  class Body *impl;
public:
  ~Handle() { delete impl; }
  // ...
};
```

In this example, the definition of class `Body` is not visible when the pointer to `Body` is deleted.

#### Correction — Define Class Before Deletion

One possible correction is to make sure that the class definition is visible when a pointer to the class is deleted.

```
class Handle {
  class Body *impl;
public:
  ~Handle();
  // ...
};

// Elsewhere
class Body { /* ... */ };

Handle::~Handle() {
  delete impl;
}
```

#### Correction — Use `std::shared_ptr`

Another possible correction is to use the `std::shared_ptr` type instead of a regular pointer.

```
#include <memory>

class Handle {
  std::shared_ptr<class Body> impl;
  public:
    Handle();
    ~Handle() {}
    // ...
};
```

### Downcasting to Pointer to Incomplete Class

`File1.h`:

```
class Base {
protected:
  double var;
public:
  Base() : var(1.0) {}
  virtual void do_something();
  virtual ~Base();
};
```

`File2.h`:

```
void funcprint(class Derived *);
class Base *get_derived();
```

`File1.cpp`:

```
#include "File1.h"
#include "File2.h"

void getandprint() {
  Base *v = get_derived();
  funcprint(reinterpret_cast<class Derived *>(v));
}
```

File2.cpp:

```
#include "File2.h"
#include "File1.h"
#include <iostream>

class Base2 {
protected:
  short var2;
public:
  Base2() : var2(12) {}
};

class Derived : public Base2, public Base {
  float var_derived;
public:
    Derived() : Base2(), Base(), var_derived(1.2f) {}
    void do_something()
    {
        std::cout << "var_derived: "
                  << var_derived << ", var : " << var
                  << ", var2: " << var2 << std::endl;
    }
 };

void funcprint(Derived *d) {
  d->do_something();
}

Base *get_derived() {
  return new Derived;
}
```

In this example, the definition of class Derived is not visible in File1.cpp when a Base* pointer to downcast to a Derived* pointer.

In File2.cpp, class Derived derives from two classes, Base and Base2. This information about multiple inheritance is not available at the point of downcasting in File1.cpp. The result of downcasting is passed to the function funcprint and dereferenced in the body of funcprint. Because the downcasting was done with incomplete information, the dereference can be invalid.

**Correction — Define Class Before Downcasting**

One possible correction is to define the class Derived before downcasting a Base* pointer to a Derived* pointer.

In this corrected example, the downcasting is done in File2.cpp in the body of funcprint at a point where the definition of class Derived is visible. The downcasting is not done in File1.cpp where the definition of Derived is not visible. The changes from the previous incorrect example are highlighted.

File1_corr.h:

```
class Base {
protected:
  double var;
public:
  Base() : var(1.0) {}
  virtual void do_something();
  virtual ~Base();
};
```

File2_corr.h:

```
void funcprint(class Base *);
class Base *get_derived();
```

File1.cpp:

```
#include "File1_corr.h"
#include "File2_corr.h"

void getandprint() {
  Base *v = get_derived();
  funcprint(v);
}
```

File2.cpp:

```
#include "File2_corr.h"
#include "File1_corr.h"
#include <iostream>

class Base2 {
protected:
  short var2;
public:
  Base2() : var2(12) {}
};

class Derived : public Base2, public Base {
  float var_derived;

public:
    Derived() : Base2(), Base(), var_derived(1.2f) {}
    void do_something()
    {
        std::cout << "var_derived: "
                  << var_derived << ", var : " << var
                  << ", var2: " << var2 << std::endl;
    }
};

void funcprint(Base *d) {
  Derived *temp = dynamic_cast<Derived*>(d);
  if(temp)  {
     d->do_something();
  }
  else {
      //Handle error
  }
}

Base *get_derived() {
  return new Derived;
}
```

## Result Information

**Group:** Object Oriented
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `INCOMPLETE_CLASS_PTR`
**Impact:** Medium

# Version History

**Introduced in R2018b**

## See Also

`Delete of void pointer` | `MISRA C++:2008 Rule 5-2-4` | `MISRA C++:2008 Rule 5-2-7` | `MISRA C++:2008 Rule 5-2-8` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Copy constructor not called in initialization list

Copy constructor does not call copy constructors of some members or base classes

## Description

This defect occurs when the copy constructor of a class does not call the *copy constructor* of the following in its initialization list:

- One or more of its members.
- Its base classes when applicable.

  The defect occurs even when a base class constructor is called instead of the base class copy constructor.

### Risk

The calls to the copy constructors can be done only from the initialization list. If the calls are missing, it is possible that an object is only partially copied.

- If the copy constructor of a member is not called, it is possible that the member is not copied.
- If the copy constructor of a base class is not called, it is possible that the base class members are not copied.

### Fix

If you want your copy constructor to perform a complete copy, call the copy constructor of all members and all base classes in its initialization list.

## Examples

### Base Class Copy Constructor Not Called

```
class Base {
public:
    Base();
    Base(int);
    Base(const Base&);
    virtual ~Base();
private:
    int ib;
};

class Derived:public Base {
public:
    Derived();
    ~Derived();
    Derived(const Derived& d): Base(), i(d.i) { }
private:
    int i;
};
```

In this example, the copy constructor of class `Derived` calls the default constructor, but not the copy constructor of class `Base`.

The defect appears on the `:` symbol in the copy constructor definition. Following are some tips for navigating in the source code:

- To navigate to the class definition, right-click a member that is initialized in the constructor. Select **Go To Definition**. In the class definition, you see the class members, including those members whose copy constructors are not called.
- To navigate to a base class definition, first navigate to the derived class definition. In the derived class definition, where the derived class inherits from a base class, right-click the base class name and select **Go To Definition**.

#### Correction — Call Base Class Copy Constructor

One possible correction is to call the copy constructor of class `Base` from the initialization list of the `Derived` class copy constructor.

```
class Base {
public:
    Base();
    Base(int);
    Base(const Base&);
    virtual ~Base();
private:
    int ib;
};

class Derived:public Base {
public:
    Derived();
    ~Derived();
    Derived(const Derived& d): Base(d), i(d.i) { }
private:
    int i;
};
```

## Result Information

**Group:** Object oriented
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `MISSING_COPY_CTOR_CALL`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)` | `Base class assignment operator not called`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"

“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Copy operation modifying source operand

Copy operation modifies data member of source object

## Description

This defect occurs when a copy constructor or copy assignment operator modifies a mutable data member of its source operand.

For instance, this copy constructor `A` modifies the data member `m` of its source operand `other`:

```
class A {
  mutable int m;

public:
 ...
  A(const A &other) : m(other.m) {
    other.m = 0; //Modification of source
  }
}
```

### Risk

A copy operation with a copy constructor (or copy assignment operator):

```
className new_object = old_object; //Calls copy constructor of className
```

copies its source operand `old_object` to its destination operand `new_object`. After the operation, you expect the destination operand to be a copy of the unmodified source operand. If the source operand is modified during copy, this assumption is violated.

### Fix

Do not modify the source operand in the copy operation.

If you are modifying the source operand in a copy constructor to implement a move operation, use a move constructor instead. Move constructors are defined in the C++11 standard and later.

## Examples

### Copy Constructor Modifying Source

```
#include <algorithm>
#include <vector>

class A {
  mutable int m;

public:
  A() : m(0) {}
  explicit A(int m) : m(m) {}
  A(const A &other) : m(other.m) {
    other.m = 0;
  }

  A& operator=(const A &other) {
    if (&other != this) {
      m = other.m;
      other.m = 0;
    }
    return *this;
  }

  int get_m() const { return m; }
};

void f() {
  std::vector<A> v{10};
  A obj(12);
  std::fill(v.begin(), v.end(), obj);
}
```

In this example, a vector of ten objects of type A is created. The std::fill function copies an object of type A, which has a data member with value 12, to each of the ten objects. After this operation, you might expect that all ten objects in the vector have a data member with value 12.

However, the first copy modifies the data member of the source to the value 0. The remaining nine copies copy this value. After the std::fill call, the first object in the vector has a data member with value 12 and the remaining objects have data members with value 0.

#### Correction — Use Move Constructor for Modifying Source

Do not modify data members of the source operand in a copy constructor or copy assignment operator. If you want your class to have a move operation, use a move constructor instead of a copy constructor.

In this corrected example, the copy constructor and copy assignment operator of class A do not modify the data member m. A separate move constructor modifies the source operand.

```
#include <algorithm>
#include <vector>

class A {
  int m;

public:
  A() : m(0) {}
  explicit A(int m) : m(m) {}

  A(const A &other) : m(other.m) {}
  A(A &&other) : m(other.m) { other.m = 0; }

  A& operator=(const A &other) {
    if (&other != this) {
      m = other.m;
    }
    return *this;
  }
  //Move constructor
  A& operator=(A &&other) {
    m = other.m;
    other.m = 0;
    return *this;
  }

  int get_m() const { return m; }
};

void f() {
  std::vector<A> v{10};
  A obj(12);
  std::fill(v.begin(), v.end(), obj);
}
```

## Result Information

**Group:** Object Oriented
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `COPY_MODIFYING_SOURCE`
**Impact:** Medium

# Version History

**Introduced in R2018b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

### External Websites

Move constructors (C++11 and beyond)

# Incompatible types prevent overriding

Derived class method hides a `virtual` base class method instead of overriding it

## Description

This defect occurs when a derived class method has the same name and number of parameters as a `virtual` base class method but:

- The parameter lists differ in at least one parameter type.
- The parameter lists differ in the presence or absence of qualifiers such as `const`.

The derived class method hides the `virtual` base class method instead of overriding it.

### Risk

- You might inadvertently hide the base class method instead of overriding it with the derived class method.
- If the base class method is hidden and you use a derived class object to call the method with the base class parameters, the derived class method is called instead. For the parameters whose types do not match the arguments that you pass, a cast takes place if possible. Otherwise, a compilation failure occurs.

### Fix

To override a base class virtual method with a derived class method, declare the methods by using identical parameter lists. For instance, change the parameter type or add a `const` qualifier if required.

In C++11 and later, you can declare intended overriding methods in the derived class by using the specifier `override`. When you declare the derived class methods by using the specifier `override`, the compilation fails if the parameter lists of the base class method and the derived class method are different. The derived class methods cannot hide base class methods inadvertently and overriding of the base class virtual methods is ensured.

Otherwise, add the line `using` *`Base_class_name`*`::`*`method_name`* to the derived class declaration. You can then access the base class method using an object of the derived class.

## Examples

### `typedef` Causing Virtual Function Hiding in Derived Class

```
class Base {
public:
    Base();
    virtual ~Base();
    virtual void func(float i);
    virtual void funcp(float* i);
    virtual void funcr(float& i);
};

typedef double Float;

class Derived: public Base {
public:
    Derived();
    ~Derived();
    void func(Float i);
    void funcp(Float* i);
    void funcr(Float& i);
};
```

In this example, because of the statement `typedef double Float;`, the `Derived` class methods `func`, `funcp`, and `funcr` have `double` arguments while the `Base` class methods with the same name have `float` arguments. Therefore, you cannot access the `Base` class methods using a `Derived` class object.

The defect appears on the method that hides a base class method. To find which base class method is hidden:

1. Navigate to the base class definition. On the **Source** pane, right-click the base class name and select **Go To Definition**.
2. In the base class definition, identify the `virtual` method that has the same name as the derived class method name.

**Correction — Unhide Base Class Method**

One possible correction is to use the same argument type for the base and derived class methods to enable overriding. Otherwise, if you want to call the `Base` class methods with the `float` arguments using a `Derived` class object, add the line `using Base::`*`method_name`* to the `Derived` class declaration.

```
class Base {
public:
    Base();
    virtual ~Base();
    virtual void func(float i);
    virtual void funcp(float* i);
    virtual void funcr(float& i);
};

typedef double Float;

class Derived: public Base {
public:
    Derived();
    ~Derived();
    using Base::func;
    using Base::funcp;
    using Base::funcr;
    void func(Float i);
    void funcp(Float* i);
    void funcr(Float& i);
};
```

#### Correction — Specify Derived Class Method by using override

Another correction is to explicitly specify the derived class methods as overriding methods by using the specifier `override`. This way, it is clear that you intend to override the base class methods in the derived class. If the overriding methods have different parameter lists than their base class counterparts, the code does not compile. As a result, the derived class methods cannot hide the base class methods.

```
class Base {
public:
    Base();
    virtual ~Base();
    virtual void func(float i);
    virtual void funcp(float* i);
    virtual void funcr(float& i);
};

typedef double Float;

class Derived: public Base {
public:
    Derived();
    ~Derived();
//    Compilation error
//  void func(Float i) override;
//  void funcp(Float* i) override;
//  void funcr(Float& i) override;

    void func(float i) override;
    void funcp(float* i) override;
    void funcr(float& i) override;
};
```

The commented out method definitions have different parameter lists compared to their base class counterparts. Because the derived class methods are declared by using the specifier `override`, the differing parameter lists do not hide the base class methods. Instead, the code fails to compile. Using the `override` specifier enforces the rule that virtual methods in base and derived classes must have identical parameter lists.

### const Qualifier Missing in Derived Class Method

```
namespace Missing_Const {
class Base {
public:
    virtual void func(int) const ;
    virtual ~Base() ;
} ;

class Derived : public Base {
public:
    virtual void func(int) ;

} ;
}
```

In this example, `Derived::func` does not have a `const` qualifier but `Base::func` does. Therefore, `Derived::func` does not override `Base::func`.

#### Correction — Add const Qualifier to Derived Class Method

To enable overriding, add the `const` qualifier to the derived class method declaration.

```
namespace Missing_Const {
class Base {
public:
    virtual void func(int) const ;
    virtual ~Base() ;
} ;

class Derived : public Base {
public:
    virtual void func(int) const;

} ;
}
```

To avoid hiding base class methods or turning virtual methods into nonvirtual methods unintentionally:

- Declare virtual methods in the base class by using the specifier `virtual`.
- Declare virtual methods in a nonfinal derived base class by using the specifier `override`.
- Declare virtual methods in the final class by using the specifier `final`.

### Value Instead of Reference in Derived Class Method

```
namespace Missing_Ref {

class Obj {
    int data;
};

class Base {
public:
    virtual void func(Obj& o);
    virtual ~Base() ;
} ;

class Derived : public Base {
public:
    virtual void func(Obj o) ;

} ;
}
```

In this example, `Derived::func` accepts an `Obj` parameter by value but `Base::func` accepts an `Obj` parameter by reference. Therefore, `Derived::func` does not override `Base::func`.

#### Correction — Use Reference for Parameter of Derived Class Method

To enable overriding, pass the derived class method parameter by reference.

```
namespace Missing_Ref {

class Obj {
    int data;
};
class Base {
public:
    virtual void func(Obj& o);
    virtual ~Base() ;
} ;

class Derived : public Base {
public:
    virtual void func(Obj& o) ;

} ;
}
```

To avoid hiding base class methods or turning virtual methods into nonvirtual methods unintentionally:

- Declare virtual methods in the base class by using the specifier `virtual`.
- Declare virtual methods in a nonfinal derived base class by using the specifier `override`.
- Declare virtual methods in the final class by using the specifier `final`.

### Result Information

**Group:** Object oriented
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `VIRTUAL_FUNC_HIDING`
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Find defects (-checkers)`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Lambda used as typeid operand

typeid is used on lambda expression

## Description

This defect occurs when you use typeid on a lambda expression.

### Risk

According to the C++ Standard, the type of a lambda expression is a unique, unnamed class type. Because the type is unique, another variable or expression cannot have the same type. Use of typeid on a lambda expression indicates that you expect a second variable or expression to have the same type as the operand lambda expression. Using the type of a lambda expression in this way can lead to unexpected results.

typeid returns the data type of its operand. Typically the operator is used to compare the types of two variables. For instance:

```
(typeid(var1) == typeid(var2))
```

compares the types of var1 and var2. This use does not apply to a lambda expression, which has a unique type.

### Fix

Avoid using the typeid operator on lambda expressions.

## Examples

### Use of typeid on Lambda Expressions

```
#include <cstdint>
#include <typeinfo>

 void func()
 {
 auto lambdaFirst = []() -> std::int8_t { return 1; };
 auto lambdaSecond = []() -> std::int8_t { return 1; };

 if (typeid(lambdaFirst) == typeid(lambdaSecond))
     {
     // ...
     }
 }
```

The use of typeid on lambda expressions can lead to unexpected results. The comparison above is false even though lambdaFirst and lambdaSecond appear to have the same body.

#### Correction – Assign Lambda Expression to Function Object Before Using `typeid`

One possible correction is to assign the lambda expression to a function object and then use the `typeid` operator on the function objects for comparison.

```
#include <cstdint>
#include <functional>
#include <typeinfo>

 void func()
 {
 std::function<std::int8_t()> functionFirst = []() { return 1; };
 std::function<std::int8_t()> functionSecond = []() { return 1; };

 if (typeid(functionFirst) == typeid(functionSecond))
     {
     // ...
     }
 }
```

## Result Information

**Group:** Object Oriented
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `LAMBDA_TYPE_MISUSE`
**Impact:** Low

# Version History

**Introduced in R2019b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Member not initialized in constructor

Constructor does not initialize some members of a class

## Description

This defect occurs when a class constructor has at least one execution path on which it does not initialize some data members of the class.

The defect does not appear in the following cases:

- Empty constructors.
- The non-initialized member is not used in the code.

### Risk

The members that the constructor does not initialize can have unintended values when you read them later.

Initializing all members in the constructor makes it easier to use your class. If you call a separate method to initialize your members and then read them, you can avoid uninitialized values. However, someone else using your class can read a class member *before* calling your initialization method. Because a constructor is called when you create an object of the class, if you initialize all members in the constructor, they cannot have uninitialized values later on.

### Fix

The best practice is to initialize all members in your constructor, preferably in an initialization list.

## Examples

### Non-Initialized Member

```
class MyClass {
public:
    explicit MyClass(int);
private:
    int _i;
    char _c;
};

MyClass::MyClass(int flag) {
    if(flag == 0) {
        _i = 0;
        _c = 'a';
    }
    else {
        _i = 1;
    }
}
```

In this example, if `flag` is not 0, the member `_c` is not initialized.

The defect appears on the closing brace of the constructor. Following are some tips for navigating in the source code:

- On the **Result Details** pane, see which members are not initialized.
- To navigate to the class definition, right-click a member that is initialized in the constructor. Select **Go To Definition**. In the class definition, you can see all the members, including those members that are not initialized in the constructor.

#### Correction — Initialize All Members on All Execution Paths

One possible correction is to initialize all members of the class `MyClass` for all values of `flag`.

```
class MyClass {
public:
    explicit MyClass(int);
private:
    int _i;
    char _c;
};

MyClass::MyClass(int flag) {
    if(flag == 0) {
        _i = 0;
        _c = 'a';
    }
    else {
        _i = 1;
        _c = 'b';
    }
}
```

## Result Information

**Group:** Object oriented
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `NON_INIT_MEMBER`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)` | `Copy constructor not called in initialization list`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Missing explicit keyword

Constructor or user-defined conversion operator missing the `explicit` specifier

## Description

This defect occurs when the declaration or in-class definition of a constructor or user-defined conversion operator does not use the `explicit` specifier. The `explicit` specifier prevents implicit conversion from a variable of another type to the current class type.

The defect applies to:

- One-parameter constructors.
- Constructors where all but one parameters have default values.

  For instance, `MyClass::MyClass(float f, bool b=true){}`.
- User-defined conversation operators.

  For instance, `operator int() {}` converts a variable of the current class type to an `int` variable.

### Risk

If you do not declare a constructor or conversion operator `explicit`, compilers can perform implicit and often unintended type conversions to or from the class type with possibly unexpected results.

The implicit conversion using a constructor can occur, for instance, when a function accepts a parameter of the class type but you call the function with an argument of a different type. The call to `func` here causes an implicit conversion from type `int` to `myClass`:

```
class myClass {}{
  ...
  myClass(int) {...}
};
void func(myClass);
func(0);
```

The reverse implicit conversion can occur when using a user-defined conversion operator. For instance, you pass the class type as argument but the function has a parameter of a different type. The call to `func` here causes an implicit conversion from type `myClass` to `int`:

```
class myClass {} {
  ...
  operator int() {...}
};
myClass myClassObject;

void func(int) {...}
func(myClassObject);
```

#### Fix

For better readability of your code and to prevent implicit conversions, in the declaration or in-class definition of the constructor or conversion operator, place the `explicit` keyword before the constructor or operator name. You can then detect all implicit conversions as compilation errors and convert them to explicit conversions.

## Examples

### Missing `explicit` Keyword on Constructor

```
class MyClass {
public:
    MyClass(int val);
private:
    int val;
};

void func(MyClass);

void main() {
    MyClass MyClassObject(0);

    func(MyClassObject);   // No conversion
    func(MyClass(0));      // Explicit conversion
    func(0);               // Implicit conversion
}
```

In this example, the constructor of `MyClass` is not declared `explicit`. Therefore, the call `func(0)` can perform an implicit conversion from `int` to `MyClass`.

#### Correction — Use explicit Keyword

One possible correction is to declare the constructor of `MyClass` as `explicit`. If an operation in your code performs an implicit conversion, the compiler generates an error. Therefore, using the `explicit` keyword, you detect unintended type conversions in the compilation stage.

For instance, in function `main` below, if you add the statement `func(0);` that performs implicit conversion, the code does not compile.

```
class MyClass {
public:
    explicit MyClass(int val);
private:
    int val;
};

void func(MyClass);

void main() {
    MyClass MyClassObject(0);

    func(MyClassObject);   // No conversion
    func(MyClass(0));      // Explicit conversion
}
```

### Incorrect Argument Order Preventable Through `explicit` Keyword

```
class Month {
    int val;
public:
    Month(int m): val(m) {}
    ~Month() {}
};

class Day {
    int val;
public:
    Day(int d): val(d) {}
    ~Day() {}
};

class Year {
    int val;
public:
    Year(int y): val(y) {}
    ~Year() {}
};

class Date {
    Month mm;
    Day dd;
    Year yyyy;
public:
    Date(const Month & m, const Day & d, const Year & y):mm(m), dd(d), yyyy(y) {}
};

void main() {
    Date(20,1,2000); //Implicit conversion, wrong argument order undetected
}
```

In this example, the constructors for classes `Month`, `Day` and `Year` do not have an `explicit` keyword. They allow implicit conversion from `int` variables to `Month`, `Day` and `Year` variables.

When you create a `Date` variable and use an incorrect argument order for the `Date` constructor, because of the implicit conversion, your code compiles. You might not detect that you have switched the month value and the day value.

#### Correction — Use explicit Keyword

If you use the `explicit` keyword for the constructors of classes `Month`, `Day` and `Year`, you cannot call the `Date` constructor with an incorrect argument order.

- If you call the `Date` constructor with `int` variables, your code does not compile because the `explicit` keyword prevents implicit conversion from `int` variables.
- If you call the `Date` constructor with the arguments explicitly converted to `Month`, `Day` and `Year`, and have the wrong argument order, your code does not compile because of the argument type mismatch.

```
class Month {
    int val;
public:
    explicit Month(int m): val(m) {}
    ~Month() {}
};

class Day {
    int val;
public:
    explicit Day(int d): val(d) {}
    ~Day() {}
};

class Year {
    int val;
public:
    explicit Year(int y): val(y) {}
    ~Year() {}
};

class Date {
    Month mm;
    Day dd;
    Year yyyy;
public:
    Date(const Month & m, const Day & d, const Year & y):mm(m), dd(d), yyyy(y) {}
};

void main() {
    Date(Month(1),Day(20),Year(2000));
    // Date(20,1,2000); - Does not compile
    // Date(Day(20), Month(1), Year(2000)); - Does not compile
}
```

### Missing `explicit` Keyword on Conversion Operator

```
#include <cstdint>

class MyClass {
public:
    explicit MyClass(int32_t arg): val(arg) {};
    operator int32_t() const { return val; }
    explicit operator bool() const {
        if (val>0) {
          return true;
        }
        return false;
     }
private:
    int32_t val;
};

void useIntVal(int32_t);
void useBoolVal(bool);

void func() {
    MyClass MyClassObject{0};
    useIntVal(MyClassObject);
    useBoolVal(static_cast<bool>(MyClassObject));
}
```

In this example, the conversion operator `operator int32_t()` is not defined with the `explicit` specifier and allows implicit conversions. The conversion operator `operator bool()` is defined explicit.

When converting to a `bool` variable, for instance, in the call to `useBoolVal`, the `explicit` keyword in the conversion operator ensures that you have to perform an explicit conversion from the type `MyClass` to `bool`. There is no such requirement when converting to an `int32_t` variable. In the call to `useIntVal`, an implicit conversion is performed.

### Result Information

**Group:** Object oriented
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `MISSING_EXPLICIT_KEYWORD`
**Impact:** Low

## Version History

**Introduced in R2015b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing virtual inheritance

A base class is inherited virtually and nonvirtually in the same hierarchy

## Description

This defect occurs when:

- A class is derived from multiple base classes, and some of those base classes are themselves derived from a common base class.

  For instance, a class `Final` is derived from two classes, `Intermediate_left` and `Intermediate_right`. Both `Intermediate_left` and `Intermediate_right` are derived from a common class, `Base`.
- At least one of the inheritances from the common base class is `virtual` and at least one is not `virtual`.

  For instance, the inheritance of `Intermediate_right` from `Base` is `virtual`. The inheritance of `Intermediate_left` from `Base` is not `virtual`.

### Risk

If this defect appears, multiple copies of the base class data members appear in the final derived class object. To access the correct copy of the base class data member, you have to qualify the member and method name appropriately in the final derived class. The development is error-prone.

For instance, when the defect occurs, two copies of the base class data members appear in an object of class `Final`. If you do not qualify method names appropriately in the class `Final`, you can assign a value to a `Base` data member but not retrieve the same value.

- You assign the value using a `Base` method accessed through `Intermediate_left`. Therefore, you assign the value to one copy of the `Base` member.
- You retrieve the value using a `Base` method accessed through `Intermediate_right`. Therefore, you retrieve a different copy of the `Base` member.

### Fix

Declare all the intermediate inheritances as `virtual` when a class is derived from multiple base classes that are themselves derived from a common base class.

If you indeed want multiple copies of the `Base` data members as represented in the intermediate derived classes, use aggregation instead of inheritance. For instance, declare two objects of class `Intermediate_left` and `Intermediate_right` in the `Final` class.

## Examples

### Missing Virtual Inheritance

```
#include <stdio.h>
class Base {
```

```
public:
    explicit Base(int i): m_b(i) {};
    virtual ~Base() {};
    virtual int get() const {
        return m_b;
    }
    virtual void set(int b) {
        m_b = b;
    }
private:
    int m_b;
};

class Intermediate_left: virtual public Base {
public:
    Intermediate_left():Base(0), m_d1(0) {};
private:
    int m_d1;
};

class Intermediate_right: public Base {
public:
    Intermediate_right():Base(0), m_d2(0) {};
private:
    int m_d2;
};

class Final: public Intermediate_left, Intermediate_right {
public:
    Final(): Base(0), Intermediate_left(), Intermediate_right() {};
    int get() const {
        return Intermediate_left::get();
    }
    void set(int b) {
        Intermediate_right::set(b);
    }
    int get2() const {
        return Intermediate_right::get();
    }
};

int main(int argc, char* argv[]) {
    Final d;
    int val = 12;
    d.set(val);
    int res = d.get();
    printf("d.get=%d\n",res);            // Result: d.get=0
    printf("d.get2=%d\n",d.get2());      // Result: d.get2=12
    return res;
}
```

In this example, `Final` is derived from both `Intermediate_left` and `Intermediate_right`. `Intermediate_left` is derived from `Base` in a non-`virtual` manner and `Intermediate_right` is derived from `Base` in a `virtual` manner. Therefore, two copies of the base class and the data member `m_b` are present in the final derived class, Both derived classes Intermediate_left and Intermediate_right do not override the Base class methods get and set. However, Final overrides both methods. In the overridden get method, it calls Base::get through Intermediate_left. In the overridden set method, it calls Base::set through Intermediate_right.

Following the statement d.set(val), Intermediate_right's copy of m_b is set to 12. However, Intermediate_left's copy of m_b is still zero. Therefore, when you call d.get(), you obtain a value zero.

Using the printf statements, you can see that you retrieve a value that is different from the value that you set.

The defect appears in the final derived class definition and on the name of the class that are derived virtually from the common base class. Following are some tips for navigating in the source code:

- To find the definition of a class, on the **Source** pane, right-click the class name and select **Go To Definition**.
- To navigate up the class hierarchy, first navigate to the intermediate class definition. In the intermediate class definition, right-click a base class name and select **Go To Definition**.

#### Correction — Make Both Inheritances Virtual

One possible correction is to declare both the inheritances from Base as virtual.

Even though the overridden get and set methods in Final still call Base::get and Base::set through different classes, only one copy of m_b exists in Final.

```
#include <stdio.h>
class Base {
public:
    explicit Base(int i): m_b(i) {};
    virtual ~Base() {};
    virtual int get() const {
        return m_b;
    }
    virtual void set(int b) {
        m_b = b;
    }
private:
    int m_b;
};

class Intermediate_left: virtual public Base {
public:
    Intermediate_left():Base(0), m_d1(0) {};
private:
    int m_d1;
};

class Intermediate_right: virtual public Base {
public:
    Intermediate_right():Base(0), m_d2(0) {};
private:
    int m_d2;
};

class Final: public Intermediate_left, Intermediate_right {
```

```
public:
    Final(): Base(0), Intermediate_left(), Intermediate_right() {};
    int get() const {
        return Intermediate_left::get();
    }
    void set(int b) {
        Intermediate_right::set(b);
    }
    int get2() const {
        return Intermediate_right::get();
    }
};

int main(int argc, char* argv[]) {
    Final d;
    int val = 12;
    d.set(val);
    int res = d.get();
    printf("d.get=%d\n",res);             // Result: d.get=12
    printf("d.get2=%d\n",d.get2());       // Result: d.get2=12
    return res;
}
```

## Result Information

**Group:** Object oriented
**Language:** C++
**Default:** Off
**Command-Line Syntax:** MISSING_VIRTUAL_INHERITANCE
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

Find defects (-checkers)

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Object slicing

Derived class object passed by value to function with base class parameter

## Description

This defect occurs when you pass a derived class object by value to a function, but the function expects a base class object as parameter.

### Risk

If you pass a derived class object *by value* to a function, you expect the derived class copy constructor to be called. If the function expects a base class object as parameter:

1. The base class copy constructor is called.
2. In the function body, the parameter is considered as a base class object.

In C++, `virtual` methods of a class are resolved at run time according to the actual type of the object. Because of object slicing, an incorrect implementation of a `virtual` method can be called. For instance, the base class contains a `virtual` method and the derived class contains an implementation of that method. When you call the `virtual` method from the function body, the base class method is called, even though you pass a derived class object to the function.

### Fix

One possible fix is to pass the object by reference or pointer. Passing by reference or pointer does not cause invocation of copy constructors. If you do not want the object to be modified, use a `const` qualifier with your function parameter.

Another possible fix is to overload the function with another function that accepts the derived class object as parameter.

## Examples

### Function Call Causing Object Slicing

```
#include <iostream>

class Base {
public:
    explicit Base(int b) {
        _b = b;
    }
    virtual ~Base() {}
    virtual int update() const;
protected:
    int _b;
};


class Derived: public Base {
```

```
public:
    explicit Derived(int b):Base(b) {}
    int update() const;
};

//Class methods definition

int Base::update() const {
    return (_b + 1);
}

int Derived::update() const {
    return (_b -1);
}


//Other function definitions
void funcPassByValue(const Base bObj) {
    std::cout << "Updated _b=" << bObj.update() << std::endl;
}

int main() {
    Derived dObj(0);
    funcPassByValue(dObj);       //Function call slices object
    return 0;
 }
```

In this example, the call `funcPassByValue(dObj)` results in the output `Updated _b=1` instead of the expected `Updated _b=-1`. Because `funcPassByValue` expects a `Base` object parameter, it calls the `Base` class copy constructor.

Therefore, even though you pass the `Derived` object `dObj`, the function `funcPassByValue` treats its parameter `b` as a `Base` object. It calls `Base::update()` instead of `Derived::update()`.

**Correction — Pass Object by Reference or Pointer**

One possible correction is to pass the `Derived` object `dObj` by reference or by pointer. In the following, corrected example, `funcPassByReference` and `funcPassByPointer` have the same objective as `funcPassByValue` in the preceding example. However, `funcPassByReference` expects a reference to a `Base` object and `funcPassByPointer` expects a pointer to a `Base` object.

Passing the `Derived` object `d` by a pointer or by reference does not slice the object. The calls `funcPassByReference(dObj)` and `funcPassByPointer(&dObj)` produce the expected result `Updated _b=-1`.

```
#include <iostream>

class Base {
public:
    explicit Base(int b) {
        _b = b;
    }
    virtual ~Base() {}
    virtual int update() const;
protected:
    int _b;
};
```

```
class Derived: public Base {
public:
    explicit Derived(int b):Base(b) {}
    int update() const;
};

//Class methods definition

int Base::update() const {
    return (_b + 1);
}

int Derived::update() const {
    return (_b -1);
}


//Other function definitions
void funcPassByReference(const Base& bRef) {
    std::cout << "Updated _b=" << bRef.update() << std::endl;
}

void funcPassByPointer(const Base* bPtr) {
    std::cout << "Updated _b=" << bPtr->update() << std::endl;
}

int main() {
    Derived dObj(0);
    funcPassByReference(dObj);       //Function call does not slice object
    funcPassByPointer(&dObj);       //Function call does not slice object
    return 0;
 }
```

**Note** If you pass by value, because a copy of the object is made, the original object is not modified. Passing by reference or by pointer makes the object vulnerable to modification. If you are concerned about your original object being modified, add a `const` qualifier to your function parameter, as in the preceding example.

## Result Information

**Group:** Object oriented
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `OBJECT_SLICING`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

**Topics**
“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Operator new not overloaded for possibly over-aligned class

Allocated storage might be smaller than object alignment requirement

## Description

This defect occurs when you do not adequately overload operator `new/new[]` and you use this operator to create an object with an alignment requirement specified with `alignas`. The checker raises a defect for these versions of throwing and non-throwing operator `new/new[]`.

- `void* operator new(std::size_t size)`
- `void* operator new(std::size_t size, const std::nothrow_t&)`
- `void* operator new[](std::size_t size)`
- `void* operator new[](std::size_t size, const std::nothrow_t&)`

The use of `alignas` indicates that you do not expect the default operator `new/new[]` to satisfy the alignment requirement or the object, and that the object is possibly over aligned. A type is over aligned if you use `alignas` to make the alignment requirement of the type larger than `std::max_align_t`. For instance, `foo` is over aligned in this code snippet because its alignment requirement is 32 bytes, but `std::max_align_t` has an alignment of 16 bytes in most implementations.

```
struct alignas(32) foo {
  char elems[32];
}
```

**Operator new not overloaded for possibly overaligned class** raises no defect if you do not overload the operator `new/new[]` and you use version C++17 or later of the Standard. The default operator `new/new[]` in C++17 or later supports over alignment by passing the alignment requirement as an argument of type `std::align_val_t`, for instance `void* operator new(std::size_t size, std::align_val_t alignment)`.

### Risk

The default operator `new/new[]` allocates storage with the alignment requirement of `std::align_val_t` at most. If you do not overload the operator when you create an object with over aligned type, the resulting object may be misaligned. Accessing this object might cause illegal access errors or abnormal program terminations.

### Fix

If you use version C++14 or earlier of the Standard, pass the alignment requirement of over aligned types to the operator `new/new[]` by overloading the operator.

## Examples

### Allocated Memory Is Smaller Than Alignment Requirement of Type foo

```
#include <new>
#include <cstdlib>
#include <iostream>

struct alignas(64) foo {
    char elems[32];
};

foo*  func()
{
    foo*  bar = 0x0;
    try {
        bar =  new  foo ;
    } catch (...) { return nullptr; }
    delete bar;
}
```

In this example, structure `foo` is declared with an alignment requirement of 32 bytes. When you use the default operator `new` to create object `bar`, the allocated memory for `bar` is smaller than the alignment requirement of type `foo` and `bar` might be misaligned.

#### Correction — Define Overloaded Operator new to Handle Alignment Requirement of Type foo

One possible correction, if you use C11 `stdlib.h` or POSIX-C `malloc.h`, is to define an overloaded operator `new` that uses `aligned_alloc()` or `posix_memalign()` or to obtain storage with the correct alignment.

```
#include <new>
#include <cstdlib>
#include <iostream>

struct alignas(64) foo {
    char elems[32];
    static void* operator new (size_t nbytes)
    {
        if (void* p =
                ::aligned_alloc(alignof(foo), nbytes)) {
            return p;
        }
        throw std::bad_alloc();
    }
    static void operator delete(void *p) {
        free(p);
    }
};

foo*  func()
{
    foo*  bar = 0x0;
    try {
        bar =  new  foo ;
    } catch (...) { return nullptr; }
    delete bar;
}
```

## Result Information

**Group:** Object Oriented
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `MISSING_OVERLOAD_NEW_FOR_ALIGNED_OBJ`
**Impact:** Medium

## Version History

**Introduced in R2019b**

## See Also

`Find defects (-checkers)` | `Missing overload of allocation or deallocation function`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Partial override of overloaded virtual functions

Class overrides fraction of inherited virtual functions with a given name

## Description

This defect occurs when:

- A base class has multiple `virtual` methods with the same name but different signatures (overloading).
- A class derived from the base class overrides at least one of those `virtual` methods, but not all of them.

### Risk

The `virtual` methods that the derived class does not override are hidden. You cannot call those methods using an object of the derived class.

### Fix

See if the overloads in the base class are required. If they are needed, possible solutions include:

- In your derived class, if you override one `virtual` method, override all `virtual` methods from the base class with the same name as that method.
- Otherwise, add the line `using` *`Base_class_name`*`::`*`method_name`* to the derived class declaration. In this way, you can call the base class methods using an object of the derived class.

## Examples

### Partial Override

```
class Base {
public:
    explicit Base(int b);
    virtual ~Base() {};
    virtual void set()           {
        _b = (int)0;
    };
    virtual void set(short i)    {
        _b = (int)i;
    };
    virtual void set(int i)      {
        _b = (int)i;
    };
    virtual void set(long i)     {
        _b = (int)i;
    };
    virtual void set(float i)    {
        _b = (int)i;
    };
    virtual void set(double i)   {
        _b = (int)i;
    };
private:
    int _b;
};

class Derived: public Base {
        public:
                Derived(int b, int d): Base(b), _d(d) {};
                void set(int i)    { Base::set(i); _d = (int)i; };
        private:
                int _d;
};
```

In this example, the class `Derived` overrides the function `set` that takes an `int` argument. It does not override other functions that have the same name `set` but take arguments of other types.

The defect appears on the derived class name in the derived class definition. To find which base class method is overridden:

1 Navigate to the base class definition. On the **Source** pane, right-click the base class name and select **Go To Definition**.
2 In the base class definition, identify the method that has the same name and signature as a derived class method name.

**Correction — Unhide Base Class Method**

One possible correction is add the line `using Base::set` to the `Derived` class declaration.

```
class Base {
public:
    explicit Base(int b);
    virtual ~Base() {};
    virtual void set()           {
        _b = (int)0;
    };
    virtual void set(short i)    {
        _b = (int)i;
    };
    virtual void set(int i)      {
        _b = (int)i;
    };
    virtual void set(long i)     {
        _b = (int)i;
    };
    virtual void set(float i)    {
        _b = (int)i;
    };
    virtual void set(double i)   {
        _b = (int)i;
    };
private:
    int _b;
};

class Derived: public Base {
        public:
                Derived(int b, int d): Base(b), _d(d) {};
                using Base::set;
                void set(int i)    { Base::set(i); _d = (int)i; };
        private:
                int _d;
};
```

### Result Information

**Group:** Object oriented
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `PARTIAL_OVERRIDE`
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Find defects (-checkers)`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Return of non-const handle to encapsulated data member

Method returns pointer or reference to internal member of object

## Description

This defect occurs when:

- A class method returns a handle to a data member. Handles include pointers and references.
- The method is more accessible than the data member. For instance, the method has access specifier `public`, but the data member is `private` or `protected`.

### Risk

The access specifier determines the accessibility of a class member. For instance, a class member declared with the `private` access specifier cannot be accessed outside a class. Therefore, nonmember, nonfriend functions cannot modify the member.

When a class method returns a handle to a less accessible data member, the member accessibility changes. For instance, if a `public` method returns a pointer to a `private` data member, the data member is effectively not `private` anymore. A nonmember, nonfriend function calling the `public` method can use the returned pointer to view and modify the data member.

Also, if you assign the pointer to a data member of an object to another pointer, when you delete the object, the second pointer can be left dangling. The second pointer points to the part of an object that does not exist anymore.

### Fix

One possible fix is to avoid returning a handle to a data member from a class method. Return a data member by value so that a copy of the member is returned. Modifying the copy does not change the data member.

If you must return a handle, use a `const` qualifier with the method return type so that the handle allows viewing, but not modifying, the data member.

## Examples

### Return of Pointer to `private` Data Member

```
#include <string>
#define NUM_RECORDS 100

struct Date {
    int dd;
    int mm;
    int yyyy;
};
```

```
struct Period {
    Date startDate;
    Date endDate;
};

class DataBaseEntry {
private:
    std::string employeeName;
    Period employmentPeriod;
public:
    Period* getPeriod(void);
};

Period* DataBaseEntry::getPeriod(void) {
    return &employmentPeriod;
}


void use(Period*);
void reset(Period*);

int main() {
    DataBaseEntry dataBase[NUM_RECORDS];
    Period* tempPeriod;
    for(int i=0;i < NUM_RECORDS;i++) {
        tempPeriod = dataBase[i].getPeriod();
        use(tempPeriod);
        reset(tempPeriod);
    }
    return 0;
}

void reset(Period* aPeriod) {
       aPeriod->startDate.dd = 1;
       aPeriod->startDate.mm = 1;
       aPeriod->startDate.yyyy = 2000;
}
```

In this example, `employmentPeriod` is `private` to the class `DataBaseEntry`. It is therefore immune from modification by nonmember, nonfriend functions. However, returning a pointer to `employmentPeriod` breaks this encapsulation. For instance, the nonmember function `reset` modifies the member `startDate` of `employmentPeriod`.

**Correction: Return Member by Value**

One possible correction is to return the data member `employmentPeriod` by value instead of pointer. Modifying the return value does not change the data member because the return value is a copy of the data member.

```
#include <string>
#define NUM_RECORDS 100

struct Date {
    int dd;
    int mm;
    int yyyy;
};
```

```
struct Period {
    Date startDate;
    Date endDate;
};

class DataBaseEntry {
private:
    std::string employeeName;
    Period employmentPeriod;
public:
    Period getPeriod(void);
};

Period DataBaseEntry::getPeriod(void) {
    return employmentPeriod;
}


void use(Period*);
void reset(Period*);

int main() {
    DataBaseEntry dataBase[NUM_RECORDS];
    Period tempPeriodVal;
    Period* tempPeriod;
    for(int i=0;i < NUM_RECORDS;i++) {
        tempPeriodVal = dataBase[i].getPeriod();
        tempPeriod = &tempPeriodVal;
        use(tempPeriod);
        reset(tempPeriod);
    }
    return 0;
}

void reset(Period* aPeriod) {
       aPeriod->startDate.dd = 1;
       aPeriod->startDate.mm = 1;
       aPeriod->startDate.yyyy = 2000;
}
```

### Result Information

**Group:** Object oriented
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `BREAKING_DATA_ENCAPSULATION`
**Impact:** Medium

## Version History

**Introduced in R2015b**

### See Also

`Find defects (-checkers)`

**Topics**

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Self assignment not tested in operator

Copy assignment operator does not test for self-assignment

## Description

This defect occurs when you do not test if the argument to the copy assignment operator of an object is the object itself. Polyspace does not raise this defect when you implement the copy operator by using the copy and swap idiom.

### Risk

Self-assignment causes unnecessary copying. Though it is unlikely that you assign an object to itself, because of aliasing, you or users of your class cannot always detect a self-assignment.

Self-assignment can cause subtle errors if a data member is a pointer and you allocate memory dynamically to the pointer. In your copy assignment operator, you typically perform these steps:

1. Deallocate the memory originally associated with the pointer.

   ```
   delete ptr;
   ```
2. Allocate new memory to the pointer. Initialize the new memory location with contents obtained from the operator argument.

   ```
    ptr = new ptrType(*(opArgument.ptr));
   ```

If the argument to the operator, `opArgument`, is the object itself, after your first step, the pointer data member in the operator argument, `opArgument.ptr`, is not associated with a memory location. `*opArgument.ptr` contains unpredictable values. Therefore, in the second step, you initialize the new memory location with unpredictable values.

### Fix

Test for self-assignment in the copy assignment operator of your class. Only after the test, perform the assignments in the copy assignment operator.

## Examples

### Missing Test for Self-Assignment

```
#include <algorithm>

class MyClass1 { };
class MyClass2 {
public:
    MyClass2() : p_(new MyClass1()) { }
    MyClass2(const MyClass2& f) : p_(new MyClass1(*f.p_)) { }
    ~MyClass2() {
        delete p_;
    }
    MyClass2& operator= (const MyClass2& f)
```

```
    {
        delete p_;
        p_ = new MyClass1(*f.p_);
        return *this;
    }
private:
    MyClass1* p_;
};

class MyClass3 {
public:
    MyClass3& operator=( MyClass3& other) {
        MyClass3 tmp(other);
        swap(tmp);
        return *this;
    }

    void swap(MyClass3& other) noexcept{
        std::swap(obj_, other.obj_);
    }


private:

    MyClass1* obj_;
};
```

In this example, the copy assignment operator in `MyClass2` does not test for self-assignment. If the parameter `f` is the current object, after the statement `delete p_`, the memory allocated to pointer `f.p_` is also deallocated. The statement `p_ = new MyClass1(*f.p_)` might initializes the memory location that `p_` points to with unpredictable values. Polyspace flags the copy assignment operator.

The class `MyClass3` implements the copy assignment operator by using the copy-and-swap idiom. When the parameter `other` is the current object, the operator creates a temporary copy of the current object by using the copy constructor. Then, the objects `tmp.obj_` and `this.obj_` is swapped before returning the `this` pointer. By creating a temporary object, this copy assignment operator avoids the unexpected behavior in `MyClass2::operator=`. Polyspace does not flag this copy assignment operator.

**Correction — Test for Self-Assignment**

One possible correction is to test for self-assignment in the copy assignment operator.

```
class MyClass1 { };
class MyClass2 {
public:
    MyClass2() : p_(new MyClass1()) { }
    MyClass2(const MyClass2& f) : p_(new MyClass1(*f.p_)) { }
    ~MyClass2() {
        delete p_;
    }
    MyClass2& operator= (const MyClass2& f)
    {
        if(&f != this) {
           delete p_;
           p_ = new MyClass1(*f.p_);
        }
        return *this;
    }
private:
    MyClass1* p_;
};
```

## Result Information

**Group:** Object oriented
**Language:** C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `MISSING_SELF_ASSIGN_TEST`
**Impact:** Medium

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

18

# Performance Defects

# A move operation may throw

Throwing move operations might result in STL containers using the corresponding copy operations

## Description

This defect occurs when a class explicitly declares a move constructor that is missing a `noexcept` specifier or has a `noexcept` specifier whose argument evaluates to `false`. The defect also occurs if an explicitly declared move constructor has the `throw(`*`type`*`)` exception specification (deprecated in C++11 and removed in C++17).

The checker does not raise a flag if the move constructor is implicitly declared or explicitly declared as `=default`.

### Risk

If a move operation can throw exceptions, some STL containers will use the copy operations instead and not get the performance benefits of a move operation. For instance, the implementation of the `std::vector::resize` method uses `std::move_if_noexcept` and performs a move operation for resizing a vector only if the move operation is declared `noexcept`.

### Fix

Add a `noexcept` specifier to the declaration of the move constructor.

If the move constructor contains expressions that might throw, fix those expressions. To detect violations of the `noexcept` exception specification, use the checker `Noexcept function exits with exception`.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Move Constructors Without `noexcept` Specifier

```
#include <string>

class Database {
  private:
      std::string* initEntry;
      int size;
  public:
      //Copy constructor
      Database (const Database& other);
      //Move constructor
      Database (Database&& other): initEntry{other.initEntry}, size{other.size} {
      other.initEntry = nullptr;
      other.size = 0;
    }
};
```

In this example, the `Database` move constructor does not have a `noexcept` specification.

### Correction – Add `noexcept` Specifier

Add the `noexcept` specifier to the move constructor.

```
#include <string>

class Database {
   private:
       std::string* initEntry;
       int size;
   public:
       //Copy constructor
       Database (const Database& other);
       //Move constructor
       Database (Database&& other) noexcept: initEntry{other.initEntry}, size{other.size} {
       other.initEntry = nullptr;
       other.size = 0;
       }
};
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `MOVE_OPERATION_MAY_THROW`
**Impact:** Low

# Version History

**Introduced in R2020b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Const parameter values may cause unnecessary data copies

Const parameter values may prevent a move operation resulting in a more performance-intensive copy operation

## Description

This defect occurs when `const` objects as function parameters may prevent a move operation resulting in a more performance-intensive copy operation.

The checker does not check if a move operation is possible in a given function call. The checker simply highlights `const` function parameters that have class types with a nontrivial copy operation and a move operation. You can determine for yourself if the parameter can be moved to the called function.

### Risk

If the function argument is an rvalue, the resources associated with the argument are no longer required and can be *moved* to parameters in the called function. Compilers ensure that the move operation is used in this situation since they are generally less expensive than copy operations. If you use a `const` object as function parameter, you explicitly prevent this compiler optimization.

### Fix

If you think that the parameter can be moved to the called function, remove the `const` qualifier from the flagged function parameter.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### `const` Parameter Value Preventing Move Operation

```
#include <string>

std::string getStringFromUser() {
    //Get a string of arbitrary length
}

void countWordsInString(const std::string str) {
    //Count number of words in string
}

void main() {
    std::string aString = getStringFromUser();
    std::string anotherString = getStringFromUser();

    std::string joinedString = aString + anotherString;
    countWordsInString(joinedString);
    countWordsInString(aString + anotherString);
}
```

In this example, the checker flags the `const str::string` parameter `str`. In situations where a move operation is possible, for example in the call:

```
countWordsInString(aString + anotherString);
```

the `const` parameter forces a copy operation, which can be significantly more expensive.

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `CONST_PARAMETER_VALUE`
**Impact:** Low

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)` | `Const return values may cause unnecessary data copies` | `Const rvalue reference parameter may cause unnecessary data copies`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Const return values may cause unnecessary data copies

Const return values may prevent a move operation resulting in a more performance-intensive copy operation

## Description

This defect occurs when `const` objects as return values may prevent a move operation resulting in a more performance-intensive copy operation.

The checker does not check if a move operation is possible for any calling function. The checker simply highlights `const` function return values that have class types with a nontrivial copy operation and a move operation.

### Risk

The resources associated with the function return value are no longer required and can be moved to objects in the calling function. Compilers ensure that the move operation is used in this situation since they are generally less expensive than copy operations. If you use a `const` object as return value, you explicitly prevent this compiler optimization.

In addition, the calling function can store the return value in a non-`const` object. The `const`-ness of the return value does not prevent any operation on the non-`const` object.

### Fix

Remove `const` qualifiers from function return values.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### `const` Return Value Preventing Move Operation

```
#include <string>

class stringPair {
    std::string str1;
    std::string str2;

    public:
    stringPair& operator=(const stringPair & aPair){
        if(&aPair != this) {
            str1 = aPair.str1;
            str2 = aPair.str2;
        }
        return *this;
    }
    const std::string getJoinedString(void) {
        return (str1 + str2);
    }
};
```

In this example, the `const` specifier on the return value of `getJoinedString` forces a copy operation instead of move operations.

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `CONST_RETURN_VALUE`
**Impact:** Low

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)` | `Const parameter values may cause unnecessary data copies` | `Const rvalue reference parameter may cause unnecessary data copies`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Const rvalue reference parameter may cause unnecessary data copies

The `const`-ness of an rvalue reference prevents intended move operation

## Description

This defect occurs when a function takes a `const` rvalue reference as parameter. For instance, this move constructor takes a `const` rvalue reference:

```
class aClass {
   public:
     aClass (const aClass&& anotherClass);
}
```

### Risk

The `const` nature of the rvalue reference parameter prevents the expected move operation.

For instance, this issue can happen when you write a move constructor by copy-paste from a copy constructor with a `const` parameter, for instance:

```
aClass (const aClass& anotherClass);
```

After the copy-paste, you might modify the & to && but forget to omit the `const` in the reference or the copy operations in the constructor body. In this case, the move constructor with the `const` rvalue reference compiles without errors but leads to an inefficient move constructor that actually copies the data.

### Fix

Remove the `const` qualifier from the rvalue reference parameter.

For instance, the move constructor in the preceding section can be rewritten as:

```
class aClass {
   public:
     aClass (aClass&& anotherClass);
}
```

You might also want to check the move constructor body and make sure that you are actually moving the data and not copying.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Move Constructors with `const` Rvalue Reference Parameters

```
#include <string>
```

```
#include <utility>

class Task {
    public:
       Task( const Task& ) = delete;
       Task( const Task&& other ) noexcept;
    private:
       std::string m_name;
       bool m_is_pending = false;
};
```

In this example, the move constructor has a `const` rvalue reference, which causes the defect.

The defect could have been introduced because the move constructor was created by copy-paste from the copy constructor that is deleted here.

#### Correction – Remove `const` from Parameter

Remove the `const` qualifier from the move constructor parameter to allow mutation of the parameter within the constructor body.

```
#include <string>
#include <utility>

class Task {
    public:
       Task( const Task& ) = delete;
       Task( Task&& other ) noexcept;
    private:
       std::string m_name;
       bool m_is_pending = false;
};
```

## Result Information

**Group:** Performance<br>
**Language:** C++<br>
**Default:** Off<br>
**Command-Line Syntax:** `CONST_RVALUE_REFERENCE_PARAMETER`<br>
**Impact:** Low

# Version History

**Introduced in R2021a**

## See Also

`Find defects (-checkers)` | `Const parameter values may cause unnecessary data copies` | `Const return values may cause unnecessary data copies` | `std::move called on an unmovable type`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"

"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Const `std::move` input may cause a more expensive object copy

Const `std::move` input cannot be moved and results in more expensive copy operation

## Description

This defect occurs when you use `std::move` on a `const` object, resulting in a more expensive copy operation.

The checker raises a violation only for class types with a nontrivial copy operation and a move operation.

### Risk

A `const` object cannot be modified and therefore cannot be moved. An `std::move` on a `const` object silently falls back to a copy operation without compilation errors. Your code might suffer from poorer performance without you noticing the issue.

### Fix

Remove the `const` qualifier from the object being moved.

If you want a copy operation instead, remove the redundant `std::move` call.

Note that this issue also triggers the checker `Move operation on const object`, which applies to all move operations on `const` objects irrespective of whether the class type has a move operation and a nontrivial copy operation. If you decide to justify the issue, you can use the same justification for both results.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### `std::move` on `const` Object

```
#include <string>

#include <string>

class MyClass {
public:
    void setName( const std::string& name ) {
        m_name = std::move( name );
    }
private:
    std::string m_name;
};
```

In this example, `std::move` is called on a `const` objects, `name`. Instead of a move assignment, a possibly more expensive copy assignment takes place.

#### Remove `const` Qualifiers

If you want move operations, remove the `const` qualifier from the definitions of the objects being moved.

```
#include <string>

#include <string>

class MyClass {
public:
    void setName( std::string& name ) {
        m_name = std::move ( name );
    }
private:
    std::string m_name;
};
```

### Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_STD_MOVE_CONST_OBJECT`
**Impact:** Medium

## Version History

**Introduced in R2020b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Empty destructors may cause unnecessary data copies

User-declared empty destructors prevent autogeneration of move constructors and move assignment operators

## Description

This defect occurs when a class definition contains a user-defined destructor that has an empty or `=default` implementation and does not declare both a move constructor and move assignment operator. For instance:

```
class aClass
{
public:
    ~aClass() noexcept
     {} // Empty body
};
class bClass
{
public:
    ~bClass() = default;
};
```

The destructors above are exactly the same as the compiler provided version, but they prevent automatic generation of the move operators. As a result, the class type is not movable.

An empty destructor is not flagged if:

- The destructor is private or protected.
- The destructor is declared final.
- The destructor is declared virtual and does not override a base class destructor.
- The destructor overrides a base class pure virtual destructor.
- The class has a trivial copy constructor (and therefore a copy operation is not more expensive than a move operation).

### Risk

Instances of this class might be unnecessarily copied in situations where a move operation would have been possible. Copy operations are more expensive than move operations and might impact performance.

### Fix

Try one of these solutions:

- Remove the empty destructor if possible. If a class does not have a destructor, the compiler generates a destructor, which is essentially the empty destructor that you explicitly declared.

  See also Rule of Zero.

- If you cannot remove the destructor, add an explicit move constructor and move assignment operator to the class definition. Use the `=default` syntax to clarify that the compiler definitions of move constructors and move assignment operators are used.

  ```
  class aClass
  {
  public:
       ~aClass() noexcept = default;
       aClass(aClass&& ) = default;
       aClass& operator=(aClass&& ) = default;
  };
  ```

  See also Rule of Five.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

### Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EMPTY_DESTRUCTOR_DEFINED`
**Impact:** Low

## Version History

**Introduced in R2020a**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive allocation in loop

Fixed sized memory is allocated or deallocated in a loop

## Description

This defect occurs when these conditions are met:

- The functions `malloc()`, `calloc()`, or `aligned_alloc()` is called inside a loop.
- The input for the memory allocating function is a fixed size.
- The function `free()` is then called to deallocate that same memory inside the loop.

### Risk

If the size of the memory is constant, allocating and deallocating memory in a loop is unnecessary and inefficient. Allocating the necessary memory once and deallocating the memory once at the end of all relevant operations is more efficient. Repeated allocation and deallocation of memory is inefficient and might lead to memory fragmentation and reduced performance.

### Fix

Perform the memory allocation and deallocation outside of a loop when the memory size is constant. For instance, before entering the loop, allocate the required memory. Use the memory inside the loop. After the relevant operations are completed, deallocate the memory after the loop.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Avoid Allocating Constant-Size Memory in a Loop

```
#include <stdbool.h>
#include <stdlib.h>

bool keep_working(void);
void use_buffer(void*);

void Loop() {
    while(keep_working())
    {
        void* buffer = malloc(10); // Defect
        use_buffer(buffer);
        free(buffer);
    }
}
```

In this example, a `buffer` that has a constant size `10` bytes is allocated, used, and deallocated in a `while` loop. The size of the object `buffer` does not change in the loop. Moving the memory management out of the loop might make the code more efficient. Polyspace flags the call to `malloc()`.

#### Correction — Allocate and Deallocate Constant-Size Memory Out of the Loop

To fix this defect, move the `malloc` statement above the loop and the `free` statement below the loop.

```
#include <stdbool.h>
#include <stdlib.h>

bool keep_working( void );
void use_buffer( void* );

void Loop() {
    void* buffer = malloc(10); // No Defect
    while(keep_working())
    {
        use_buffer(buffer);
    }
    free(buffer);
}
```

### Avoid Using `calloc()` with Constant Arguments in a Loop

```
#include <stdbool.h>
#include <stdlib.h>

bool keep_working(void);
void use_buffer(void*);

void Loop() {

    while(keep_working())
    {
        void* buffer = calloc(10,4); // Defect
        use_buffer(buffer);
        free(buffer);
    }
}
```

In this example, a `buffer` containing `10` elements of four bytes each is allocated, used, and deallocated in a `while` loop. The size of `buffer` does not change in the loop. Moving the memory management out of the loop might make the code more efficient. Polyspace flags the call to `calloc()`.

#### Correction — Allocate and Deallocate Constant Sized Memory Out of the Loop

To fix this defect, move the `calloc` statement above the loop and the `free` statement below the loop. Because `calloc()` allocates the memory, and then initiates all bytes to zero, the function `use_buffer()` expects a cleared buffer. To avoid incorrect behavior, reset the memory to zero at the beginning of the loop.

```
#include <stdbool.h>
#include <stdlib.h>

bool keep_working(void);
void use_buffer(void*);
void Loop() {
    void* buffer = calloc(10,5); // No Defect
    while(keep_working())
    {
        memset(buffer,0,50);//reset the memory to zero
        use_buffer(buffer);
        free(buffer);
    }
}
```

## Result Information

**Group:** Performance
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** EXPENSIVE_ALLOC_IN_LOOP
**Impact:** Medium

# Version History

**Introduced in R2022a**

## See Also

Find defects (-checkers)

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive construction of std::string or std::regex from constant string

A constant `std::string` or `std::regex` object is constructed from constant data, resulting in inefficient code

## Description

Polyspace reports this defect when both of these conditions are true:

- You construct an `std::string` or `std::regex` object from constant data, such as a string literal or the output of a `constexpr` function.
- The `std::string` or `std::regex` object remains constant or unmodified after the construction.

This defect checker does not flag class member variables, and string literals that are function arguments.

### Risk

Consider a code block that constructs an `std::string` or `std::regex` object from constant data that remains unmodified after construction. Every time this code block executes, a new object is constructed without the same constant data. Repeated construction of such an object with no modification to the content is inefficient and difficult for you detect. Consider this code:

```
#include <string>
constexpr char* getStrPtr() {
    return "abcd";
}
void foo(){
    std::string s1 = "abcd";
    std::regex reg1{"(\\w+)"};
    std::string s2 = getStrPtr();
}
int main(){
    //...
    for(int i = 0; i<10000; ++i)
    foo();
}
```

In this code, `main()` invokes the function `foo` `10000` times in a loop. Each time, the function `foo` constructs the objects `reg1`, `s1`, `s2` from the same constant string literal, resulting in inefficient code. Because such inefficient code compiles and functions correctly, you might not notice the inefficient construction of these objects from constant data.

### Fix

The fix for this defect depends on how you intend to use the constant data.

- If you do not need to reuse the constant data, use is directly as temporary literals
- If you need to reuse the constant data and you need the functionalities of `std::string` or `std::regex` class, store the constant data in a `static` string object.

- If you need to reuse the constant data and you do not need the functionalities of or `std::regex` class, store the constant data by using a `const` character array or an `std::string_view` object. Use of `std::string_view` is supported by C++17 and later.

Consider this code:

```
constexpr char* getStrPtr() {
    return "abcd";
}
void foo(){
    static const std::string s1 = getStrPtr();
    static const std::regex reg1{"(\\w+)"};
    std::string_view s1a{s1};

}
int main(){
    //...
    for(int i = 0; i<10000; ++i)
    foo();
}
```

In this code, you declare the objects `reg1` and `s1` as `static const`. Because these are `static`, they are constructed only once even if `foo` is called `10000` times. The `std::string_view` object `s1a` shows the content of `s1` and avoids the construction of an `std::string` object. By using `std::string_view` and `static const` objects, you avoid unnecessary construction of constant objects. This method also clarifies that the objects `s1` and `s1a` represent the same data.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Reconstructing Constant std::string and std::regex Objects

In this example, you declare several `std::string` and `std::regex` objects that are not modified after their construction.

- Because `foo()` constructs the objects `reg1`, `s1`, `s2`, `s5`, and `s6` from string literals and these objects remains unmodified after construction, Polyspace flags these objects.
- Because `foo()` constructs `s4` and `s3` from compile-time constants and these objects remains unmodified after construction, Polyspace flags these objects. The object `s3a` is not flagged because the output of `getCount` is not a constant.
- Polyspace does not flag these objects when they are constructed from constant data:
  - An object that is not an `std::string`, such as `*p`.
  - Temporary objects that are constructed as a function argument, such as the object containing the string literal `message` in the argument of `CallFunc`.

```
#include <string>
#include <regex>
constexpr char* getStrPtr() {
    return "abcd";
}
constexpr size_t FOUR(){
    return 4;
}
size_t getCount();
void CallFunc(std::string s);
void foo(){
    std::regex reg1{"(\\w+)"};
    std::string s1 = "abcd";
    std::string s2{"abcd"};
    std::string s3 = getStrPtr();
    std::string s4("abcd", FOUR());
    std::string s5("abcd"), s6("abcd");
}

void bar(){
    std::string s3a("abcd", getCount());
    char *p = "abcd";
    std::string s_p = p;
    CallFunc("message");
}
```

**Correction**

You can fix this defect using several enhancements:

- Declare the objects `s1`, `reg1` and `s4` as `static const`. When an object is `static`, the compiler does not reconstruct it in different scopes. When you need the functionalities of an `std::string` or `std::regex` class, this declaration is a good fix.
- Declare `s3` by using a constant character pointer `p`.
- Declare the constant strings `s2`, `s5` and `s6` as `std::string_view` objects. These objects do not contain copies of the constant strings, which makes these objects efficient.

```
#include <string>
#include <string_view>
#include<regex>
constexpr char* getStrPtr() {
    return "abcd";
}
constexpr size_t FOUR(){
    return 4;
}

void foo(){
    static const std::string s1 = "abcd";
    static const std::regex reg1{"(\\w+)"};
    std::string_view s2{s1};
    const char *p = getStrPtr();
    std::string s3 = p;
    static std::string s4("abcd", FOUR());
    std::string_view s5{s1}, s6{s4};
}
```

## Result Information

**Group:** Performance
**Language:** C++

**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_CONSTANT_STD_STRING`
**Impact:** Medium

## Version History

**Introduced in R2020b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Expensive copy in a range-based for loop iteration

The loop variable of a range-based `for` loop is copied from the range elements instead of being referenced resulting in inefficient code

## Description

This defect occurs when the loop variable of a range-based `for` loop is copied from the range elements instead of reading the range elements by reference. Copy the range elements only when its necessary because copying them might result in inefficient code. This defect is raised when the loop variable is unmodified and any of these conditions are true:

- The copied loop variable is a large trivially copyable type variable. Copying a trivially copyable object is more expensive than referencing it when the object is large.
- The copied loop variable is a nontrivially copyable type. Copying such an object might require an external function call, which is more expensive than referencing it. To check whether an object is nontrivially copyable, use the function `std::is_trivially_copyable`. For more details about this function, see `std::is_trivially_copyable` in the C++ reference.

### Risk

Range-based `for` loops can become inefficient when an expensive copy of the loop variable is made in each iteration of the loop. Consider this code:

```
void foo( std::map<std::string, std::string> const& property_map )
{
    for( std::pair< const std::string, std::string > const property: property_map)
    {}
}
```

The loop variable `property` is declared as a `const` instead of `const&`. In each iteration of the `for` loop, an `std::pair` object is copied from the map `property_maps` to the loop variable `property`. Because of the missing & in the declaration of `propert`, an expensive copy operation is done in each iteration instead of a referencing operation, resulting in inefficient code. Because this code compiles and functions correctly, the inefficient `for` loops might not be noticed. For similar source of inefficiencies, see `Expensive pass by value` and `Expensive return by value`.

### Fix

To fix this defect, declare the loop variable of a range-based `for` loop as a `const&`. Consider this code:

```
void foo( std::map<std::string, std::string> const& property_map )
{
    for( std::pair< const std::string, std::string > const& property: property_map)
    {}
}
```

Because the loop variable `property` is declared as a `const&`, the variable references a different element of the map `property_map` in each loop iteration, without copying any resource. By preventing an expensive copy in each iteration, the code becomes more efficient.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Expensive Copy in Loop Iterations

```
#include <initializer_list>
#include <unordered_map>
#include <vector>
struct Small_Trivial_Type
{
    unsigned char values[ sizeof( void* ) ];
};

struct Large_Trivial_Type
{
    unsigned char values[ 4u * sizeof( void* ) ];
};

class Nontrivial_Type
{
    Nontrivial_Type() noexcept;
    Nontrivial_Type( Nontrivial_Type const& );
    Nontrivial_Type& operator=( Nontrivial_Type const& );
    ~Nontrivial_Type() noexcept;
    int read() const;
    void modify( int );
};
extern std::vector< Nontrivial_Type > getNtts();

void foo( std::vector< Nontrivial_Type > const& ntts )
{
    for( Nontrivial_Type ntt: ntts )
    {}
}

void foo_auto( std::vector< Nontrivial_Type > const& ntts )
{
    for( auto ntt: ntts )
    {}
}
void foo_c_array( Nontrivial_Type const ( & ntts )[ 10 ] )
{
    for( Nontrivial_Type ntt: ntts )
    {}
}
void foo_large( std::vector< Large_Trivial_Type > const& ltts )
{
    for( Large_Trivial_Type ltt: ltts )
    {}
}
void foo_small( std::vector< Small_Trivial_Type > const& stts )
{
    for( Small_Trivial_Type const stt: stts )
    {}
}
```

```
void modify_elem( std::vector< Nontrivial_Type > const& ntts )
{
    for( Nontrivial_Type ntt: ntts )
    {
        ntt.modify( 42 );//Modification
    }
}
```

In this example, range-based `for` loops that have different types of loop variables are shown.

- Polyspace flags the nontrivially copyable loop variable `ntt` in `foo()` due to an expensive copy operation that is unnecessary because the loop variable is not modified. For the same reason, the loop variables in `foo_auto()` and `foo_c_array()` are flagged.
- Polyspace flags the large loop variable `ltt` in `foo_large()` because it is more expensive to copy the elements of `ltts` into `ltt` than to reference elements of `ltts`, even though `ltt` is a trivially copyable type.
- Polyspace does not flag the loop variable `stt` in `foo_small()` because copying the elements of `stts` into `stt` is not more expensive than referencing the elements of `stts`.
- Polyspace does not flag the loop variable `ntt` in `modify_elem()` because the loop variable is modified in the loop.

**Correction**

To fix this issue, use constant references (`const&`) as loop variables in range-based `for` loops. Using `const&` loop variables prevents expensive copying and produces efficient code.

```
#include <initializer_list>
#include <unordered_map>
#include <vector>
struct Small_Trivial_Type
{
    unsigned char values[ sizeof( void* ) ];
};

struct Large_Trivial_Type
{
    unsigned char values[ 4u * sizeof( void* ) ];
};

class Nontrivial_Type
{
    Nontrivial_Type() noexcept;
    Nontrivial_Type( Nontrivial_Type const& );
    Nontrivial_Type& operator=( Nontrivial_Type const& );
    ~Nontrivial_Type() noexcept;
    int read() const;
    void modify( int );
};
extern std::vector< Nontrivial_Type > getNtts();
// Test iterating over a const vector.
void foo( std::vector< Nontrivial_Type > const& ntts )
{
    for( Nontrivial_Type const& ntt: ntts ) // NC2C
    {}
}
```

```
void foo_auto( std::vector< Nontrivial_Type > const& ntts )
{
    for( auto const& ntt: ntts ) //NC2C
    {}
}
void foo_c_array( Nontrivial_Type const ( & ntts )[ 10 ] )
{
    for( Nontrivial_Type const& ntt: ntts ) // NC2C
    {}
}
void foo_large( std::vector< Large_Trivial_Type > const& ltts )
{
    for( Large_Trivial_Type const& ltt: ltts )
    {}
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_RANGE_BASED_FOR_LOOP_ITERATION`
**Impact:** Medium

# Version History

**Introduced in R2020b**

## See Also

`Find defects (-checkers)` | `Expensive pass by value` | `Expensive return by value` | `Expensive local variable copy`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive dynamic_cast

Expensive `dynamic_cast` is used instead of more efficient `static_cast` or `const_cast`

## Description

This defect is raised when `dynamic_cast` is used on a pointer, which is then immediately dereferenced. For instance:

```
std::iostream* iostream_ptr;
//...
std::string str = dynamic_cast< std::stringstream* >( iostream_ptr )->str();
```

The `iostream` pointer `iosreeam_ptr` is cast into a `stringstream` pointer, and then immediately dereferenced. Such use implies that the casting always succeeds. When you know that a casting operation succeeds, `static_cast` or `const_cast` are a more efficient choice.

### Risk

When casting one class to another in a polymorphic hierarchy, you might want to use `dynamic_cast` when run-time type of the most derived class in the hierarchy is unknown. The `dynamic_cast` is more powerful because it checks the type of the argument at run time and reports error if the check fails. These functionalities make `dynamic_cast` a more expensive operation than either of `static_cast` or `const_cast`. Because `dynamic_cast` can achieve many different goals, its use might hide the developers intent when casting. Using `dynamic_cast` when cheaper or more explicit casting operations might be more appropriate results in code that is inefficient and more difficult to maintain. Because such code compiles and runs correctly, the inefficiency might remain undetected.

### Fix

To fix this defect, replace the `dynamic_cast` with a more appropriate cheaper option. For instance:

- When calling virtual functions in a polymorphic base class, remove any casting operation.
- When downcasting from a base class to derived class, use `static_cast` if the casting operation succeeds in all conditions.
- When sidecasting from one base class to another base class, use `static_cast` if the casting operation succeeds in all conditions.
- When upcasting from a derived class to a base class, use `static_cast`.
- To modify the `const` or `volatile` qualifiers of an object, use `const_cast`.
- Refactor your code to remove inappropriate casting such as casting between unrelated classes.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Casting a Base Class into a Derived Class

```
#include <cstddef>
```

```
#include <sstream>
#include <string>
// downcast using `dynamic_cast` with pointer,
// and unconditionally call member function
void Downcast_NC( std::iostream* iostream_ptr )
{
    // ...
    std::string str = dynamic_cast< std::stringstream* >( iostream_ptr )->str();
    // ...
}
```

In this example, an `std::iostream*` object is cast into a `std::stringstream*` object by calling `dynamic_cast`. After the casting, the cast pointer is immediately dereferenced. Polyspace flags the `dynamic_cast`.

#### Correction

To fix this defect, replace `dynamic_cast` by using `static_cast`. Because `static_cast` does not check whether a casting operation might fail, consider adding an `assert` statement to check if the conversion fails. After completing the development and debugging, you might remove the `assert` statement.

```
#include <cstddef>
#include <sstream>
#include <string>
void Downcast_C( std::iostream* iostream_ptr )
// if `dynamic_cast` may fail
{
    // ...
    assert( dynamic_cast< std::stringstream* >( iostream_ptr ) != NULL );//Only for debugging
    std::string str = static_cast< std::stringstream* >( iostream_ptr )->str();
}
```

### Sidecasting and Upcasting Classes in a Polymorphic Hierarchy

```
class A{
    //...
public:
    virtual void func_A() ;
};
class B{
    //...
public:
    virtual void func_B() ;
};
class C: public A, public B{/**/};
void foo(A& a){

    dynamic_cast<B*> (&a)->func_B();//Noncompliant
}
void bar(C& c){

    dynamic_cast<B*> (&c)->func_B();//Noncompliant
}
```

In this example, the class C is derived from A and B. The function foo casts the A& object a into a B* type to access the member function B::func_B. The casting operation calls dynamic_cast, which is inefficient. Polyspace flags this conversion.

The casting from the C& object to its base class, as shown in the function bar, always succeeds. Using dynamic_cast for this operation is unnecessary and Polyspace flags the conversion.

**Correction**

Sidecasting through an unknown most derived class is a unique capability of dynamic_cast. If your code cannot function properly without a sidecast, use dynamic_cast and justify the defect. See "Annotate Code and Hide Known or Acceptable Results".

In some cases, it might be more efficient to replace a sidecast with a downcast. For instance, the function foo uses sidecasting for the explicit purpose of accessing the members of another branch in a class hierarchy. In such cases, it might be more efficient to perform the casting through the most derived class. Instead of using dynamic_cast to cast a into a B* object, you might use static_cast to cast a into a C* object. Such a downcast enables access to B::func_B while making the code more efficient. To check if the static conversion succeeds in all conditions, use assert statements during development and debugging.

Upcasting from a derived class to a base class, as shown in the function bar, always succeeds. Use static_cast for such conversions. In the preceding code, because func_B is a public virtual function, invoke the function directly by using the pointer to the class C. Casting is not necessary in this case.

```
class A{
    //...
public:
    virtual void func_A() ;
};
class B{
    //...
public:
    virtual void func_B() ;
};
class C: public A, public B{/**/};
void foo(A& a){

    static_cast<C*> (&a)->func_B();//Compliant
}
void bar(C& c){

    c.func_B();//Compliant
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** EXPENSIVE_DYNAMIC_CAST
**Impact:** Low

## Version History

**Introduced in R2021b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Expensive local variable copy

Local variable is created by copy from a `const` reference and not modified later

## Description

This defect occurs when a local variable is created by copy from a `const` reference but not modified later.

For instance, the variable `name` is created by copy from a `const` reference returned from the `get_name` function:

```
const std::string& get_name();
...
void func {
    std::string name = get_name();
}
```

The defect is raised only if the local variable has a non-trivially copyable type or a trivially copyable type with size greater than `2 * sizeof(void *)`.

### Risk

If a variable is created from a `const` reference and not modified later, the variable itself can be defined as a `const` reference. Creating a `const` reference avoids a potentially expensive copy operation.

### Fix

Avoid creating a new local variable by copy from a `const` reference if you do not intend to modify the variable later. Create a `const` reference instead.

For instance, in the preceding section, you can redefine the variable `name` as:

```
const std::string& get_name();
...
void func {
    const std::string& name = get_name();
}
```

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Expensive String Created from `const` Reference

```
#include <string>

class Task
{
public:
    // ...
    const std::string& get_name() const;
    // ...
private:
    // ...
};

void inspect( const Task& task )
{
    // ...
    const std::string name = task.get_name();
    // ...
}
```

In this example, the variable `name` is created by copy from a `const` reference but not modified later.

**Correction – Use `const` Reference**

To avoid a potentially expensive copy operation, avoid creating a new local variable if you do not intend to modify the variable later. Instead, assign the `const`-reference return value to another `const` reference.

```
#include <string>

class Task
{
public:
    // ...
    const std::string& get_name() const;
    // ...
private:
    // ...
};

void inspect( const Task& task )
{
    // ...
    const std::string& name = task.get_name();
    // ...
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_LOCAL_VARIABLE`
**Impact:** Medium

# Version History

**Introduced in R2021a**

## See Also

`Find defects (-checkers)` | `Expensive pass by value` | `Expensive return by value` | `Expensive copy in a range-based for loop iteration` | `Unmodified variable not const-qualified`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Expensive logical operation

A logical operation requires the evaluation of both operands because of their order, resulting in inefficient code

## Description

This defect occurs when all of these conditions are true:

- Left and right operands have no side effects.
- The right operand does not contain any calls to `const` member functions.
- The left operand contains one or more calls to `const` member functions.

When assessing possible side effects of an operand:

- Polyspace assumes that `const` member functions of a class do not have side effects. Nonmember functions are assumed to have side effects.
- Polyspace treats floating-point operations in accordance to the C++ standard. In C++03 or earlier, floating-point operations have no side effects. In C++11 or later, floating-point operations might have side effects, such as modifying the floating-point status flags to indicate abnormal results or auxiliary information. See Floating-point environment.
- Polyspace treats the `bool` conversion operator and logical `NOT` operators of a `struct` or a class as built-in operators. These operations are not treated as member function calls. The standard template library contains many classes that define such a `bool` conversion operator or a logical `NOT` operator.

### Risk

When evaluating logical operation, the compiler evaluates the left argument first, and then evaluates the right argument only when necessary. In a logical operation, it is inefficient to put function calls as the left argument while putting constant and variables as the right argument. Consider this code:

```
if(Object.attribute()|| var1){
//...
}
```

In the logical expression inside the `if` statement, the compiler always evaluates the function call `Object.attribute()`. Evaluating the function is not always necessary. For instance, if `var1` evaluates to `true`, then the logical expression always evaluates to true. Because `var1` is the right operand, not the left operand, the compiler unnecessarily evaluates a function call, which is inefficient. Because the inefficient code compiles and behaves correctly, this defect might go unnoticed.

### Fix

To fix this defect, flip the order of the operands in a logical expression if the left operand does not perform an operation that must be performed *before* the right operand in order to evaluate the right operand safely and correctly.

If this condition is not true, then the code relies on the exact order in which the compiler evaluates the flagged logical expression. The best practice is not to rely on the evaluation order of an expression. Consider refactoring your code so that the order of evaluation has no impact. If refactoring the code is not possible, justify the defect by using annotation or review information. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Member Function Call as Left Operand

```
#include <string>
bool updateNewProperty( const std::string& name );
void updateNewMetaProperty( const std::string& name );
volatile char externalFlag;

void updateProperty( const std::string& name )
{
    bool is_new_property = updateNewProperty( name );

    if( name.compare( "meta" ) == 0 && is_new_property )
    {
        updateNewMetaProperty( name );
    }
    if( name.compare( "meta" ) == 0 && externalFlag )
    {
        updateNewMetaProperty( name );
    }

}
```

In the first `if` statement, the variable `is_new_property` is the right operand of a logical operand. The member function `std::string::compare` is called as the left argument. The compiler evaluates the function call regardless of the value of `is_new_property`, which is inefficient. Polyspace flags the logical operation.

In the second `if` statement, the `volatile` variable `externalFlag` is the right operand. Because the variable is `volatile`, Polyspace assumes it might cause a side effect. Because the `volatile` variable might have a side effect, Polyspace does not flag the logical operation.

#### Correction

Determine if the order of the operands needs to be maintained to evaluate the expression safely and correctly. In this case, the two operands `is_new_property` and `name.compare( "meta" ) == 0` are independent and changing their order does not change the value of the logical expression. To fix this defect, use `is_new_property` as the left operand.

```
#include <string>
bool updateNewProperty( const std::string& name );
void updateNewMetaProperty( const std::string& name );
```

```
volatile char externalFlag;

void updateProperty( const std::string& name )
{
    bool is_new_property = updateNewProperty( name );
    if(is_new_property && name.compare( "meta" ) == 0 )
    {
        updateNewMetaProperty( name );
    }
    if( name.compare( "meta" ) == 0 && externalFlag )
    {
        updateNewMetaProperty( name );
    }
}
```

The compiler evaluates the call to `std::string::compare` in the first `if` statement only when `is_new_property` is `true`.

### Floating-Point Operation in Logical Expression

When you use floating-point operations in a logical expression, Polyspace estimates the side effects of the operands differently based on the version of C++ that you use. In C++03 or earlier versions, floating-point operations do not have any side effects by themselves. In C++11 or later, floating-point operations themselves might have side effects.

```
class A{
    //...
    float makeFloat() const{
        //..
    }
    void testfloat(){
        if( makeFloat() == 0.1f && fp==0.2f)
        {
            //...
        }
    }

private:
    float fp;
};
```

In this code, if you use C++03, neither of the operands has side effects. Because the left operand invokes a member function call, Polyspace flags the expression.

If you use C++11 or later, the floating-point operations might have side effects. In this case, Polyspace does not flag the logical expression.

#### Correction

Determine if the order of the operands needs to be maintained to evaluate the expression safely and correctly. In this case, the two operands `fp==0.2f` and `makeFloat() == 0.1f` are independent and changing their order does not change the value of the logical expression. To fix this defect, use `fp==0.2f` as the left operand.

```
class A{
    //...
    float makeFloat() const{
        //..
    }
    void testfloat(){
        if( fp==0.2f && makeFloat() == 0.1f)
        {
            //...
        }
    }

private:
    float fp;
};
```

The compiler evaluates the call to `makeFloat()` only when `fp==0.2f` evaluates to `true`.

### Logical Expression that Requires Specific Evaluation Order

```
#include<cstdlib>
class A{
    //...
    bool isLoaded() const { return p != NULL; }
    int get() {
        if(isLoaded() && *p > 0) {
            return *p;
        }
    }


private:
    int* p;
};
```

In the expression `(isLoaded() && *p > 0)`, the dereferencing of `*p` in the right argument is safe only when the left argument is `true`. Polyspace does not check when logical expression require such specific evaluation order. Because neither operands have side effects and the member function call is the left operand, Polyspace flags the operation.

#### Correction

In this case, the order if the operands in the logical expression needs to be maintained to evaluate the expression safely and correctly. To fix this defect, refactor your code. The best practice is not to rely on the order of evaluation of an expression.

```
#include<cstdlib>
class A{
    //...
    bool isLoaded() const { return p != NULL; }
    int get() {
        if(isLoaded()== true) {
            if(*p > 0){
                return *p;
            }

        }
    }


private:
    int* p;
};
```

This code checks the two conditions separately and does not rely on the order of evaluation. If such refactoring is not feasible, justify the defect by using annotations or review information.

### Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_LOGICAL_OPERATION`
**Impact:** Low

## Version History

**Introduced in R2021a**

### See Also

`Find defects (-checkers)` | `CERT C: Rule EXP30-C`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
List of Classes in STL with `bool` Conversion Operator

# Expensive pass by value

Parameter might be expensive to copy

## Description

This defect occurs when you pass a parameter by value instead of by reference or pointer, but the parameter is unmodified and either:

- The parameter is a non-trivially copyable type. For more on non-trivially copyable types, see *is_trivially_copyable*.
- The parameter is a trivially copyable type that is above a certain size. For example, an object greater than `2 * sizeof(void *)` is more expensive to pass by value than by reference or pointer.

Polyspace flags unmodified parameters that meet the preceding conditions even if the parameters are not declared `const`.

Polyspace raises no defect if:

- The passed by value parameter is a move-only type. For instance, `std::unique_ptr` can be moved-from but cannot be copied.
- The passed by value parameter is modified.
- The parameter is passed by value to a `virtual` overriding method in a derived class.

  In this instance, fixing the defect might require modifying the signature of the base class method which might not be possible for the owner of the derived class.
- The parameter is passed by value to a `virtual` method in a base class and the parameter is non-`const`.

  In this instance, if there are overriding methods in derived classes that modify their parameter, it might not be possible to properly update those methods if a fix is applied to the base class method.

  Note that if the parameter is `const`, overriding methods in derived classes do not modify their parameters and applying a fix in the base class is safe. In this case, Polyspace reports an expensive pass by value defect.

For example, in the following code, Polyspace does not report a defect in these functions and methods:

- The function `offset()` because its parameter is modified.
- The function `moveOnly()` because its parameter is a move-only type.
- Methods `D::parse_M` and `D::print_M` because they are `virtual` overriding methods in a derived class.
- The method `Base::print_M` because it is a `virtual` method in a base class with a non-`const` parameter.

Polyspace does report a defect for `Base::parse_M` because its parameter is `const` and non-trivially copyable.

```
#include <string>
#include <memory>
#include <iostream>

typedef struct Buffer
{
    unsigned char bytes[20];
    int index;
} Buffer;

void offset(Buffer modifiedBuffer)
{
    ++modifiedBuffer.index;
}

void moveOnly(std::unique_ptr<Buffer> move_only_param);

// virtual methods example

class Base
{
public:
    virtual void parse_M(const std::string msg);
    virtual void print_M(std::string msg)
    {
        std::cout << "Base message: " << msg << "\n";
    }
};

class D : public Base
{
public:
    void parse_M(const std::string msg) override;
    void print_M(std::string msg) override
    {
        std::cout << "Derived message: " << msg << "\n";
    }
};
```

### Risk

Passing a parameter by value creates a copy of the parameter which is inefficient if the parameter is expensive to copy. Even if your intent is to pass by reference or pointer, you might forget the `const&` or `const*` in your function signature and inadvertently run an inefficient version of the function.

### Fix

Convert the parameter to a `const` pointer (`const*`) for C code, or to a `const` reference (`const&`) for C++ code.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Expensive Pass by Value in C++ Setter Function

```
#include<string>

class Player
{
public:
    void setName(std::string const str)
    {
        name = str;
    }
    void setRank(size_t r)
    {
        rank = r;
    }
    // getter functions implementation
private:
    std::string name;
    size_t rank;

};
```

In this example, Polyspace flags the parameter of setter function `setName` which is passed by value and results in an expensive copy. The type `std::string` is not trivially copyable. The passed by value parameter of `setRank` is not flagged because `size_t` is a small trivially copyable type.

#### Correction — Pass `std::string` Parameter by `const` Reference

To avoid an inefficient copy operation, use a `const&` to pass the parameter.

```
#include<string>

class Player
{
public:
    void set_name(std::string const& s)
    {
        name = s;
    }
    void set_rank(size_t r)
    {
        rank = r;
    }
    // getter functions implementation
private:
    std::string name;
    size_t rank;

};
```

### Expensive Pass by Value in C Function

```
#include<stdio.h>
#include<string.h>

typedef struct _Player {
    char name[50];
    size_t rank;
} Player;


void printPlayer(Player const player)
{

    printf("Player name: %s\n", player.name);
    printf("Player rank: %zu\n", player.rank);
}
```

In this example, Polyspace flags the parameter of `printPlayer` which is passed by value and results in an expensive copy.

#### Correction — Pass Large Struct by `const` Pointer

To avoid an inefficient copy operation, use a `const*` to pass the parameter, then use the appropriate notation to read the structure elements.

```
#include<stdio.h>
#include<string.h>

typedef struct _Player {
    char name[50];
    size_t rank;
} Player;


void printPlayer(Player const* player)
{

    printf("Player name: %s\n", player->name);
    printf("Player rank: %zu\n", player->rank);
}
```

## Result Information

**Group:** Performance
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_PASS_BY_VALUE`
**Impact:** Medium

# Version History

**Introduced in R2020b**

## See Also

`Find defects (-checkers)` | `Expensive return by value` | `Expensive copy in a range-based for loop iteration` | `Expensive local variable copy`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"

“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Expensive post-increment operation

Object is post-incremented when pre-incrementing is faster

## Description

This defect is raised when you use the post-increment or post-decrement operation instead of a more efficient pre-increment or pre-decrement operation. The pre-increment operation is equally or more efficient when all of these conditions are true:

- Pre and post increment or decrement operations are defined for the object.
- The return type of post-increment or decrement operation is expensive to copy.
- The returned value is unused.
- The return type of pre-increment or decrement operation is not expensive to copy, such as a reference.

The efficiencies of pre-increment and pre-decrement operations depends on the version and implementation of the C++ standard that you use. If you switch C++ version or the library implementation, you might see a change in the number of violation of this check.

### Risk

Post-increment or decrement operations create a copy of the object, increment or decrement the original, and then return the copied object. When the object is expensive to copy and you do not use the returned copy of the object, the post-increment or post-decrement operation is inefficient. Use a pre-increment or pre-decrement operation, which does not copy the object and typically returns a reference to the incremented or decrement object. Inadvertently using post-increment instead of pre-increment with a large object might make the code inefficient. Code that uses inefficient post-increment or post-decrement operations compiles and behaves correctly. The inefficient operations might remain undetected.

### Fix

When you do not use the returned object from a decrement or increment operation, use a pre-increment operation.

When iterating over each element of a container, you might want to use loops that do not require increment or decrement operations, such as `std::for_each` or the range-based `for` loop. These loops are optimized for such iterations. They do not require manual increment or decrement operations.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Avoid Using Post-Increment Operation Instead of Pre-Increment Operation

```
// Using C++ 03
```

```
#include <complex>
#include <iterator> // istream_iterator
#include <sstream> // istringstream
#include <string>
#include <vector>

std::vector< std::complex< double > > deserialize( const std::string& s )
{
    std::vector< std::complex< double > > v;
    std::stringstream iss( s );
    for( std::istream_iterator< std::complex< double > > it( iss );
         it != std::istream_iterator< std::complex< double > >(); it++ )
    { v.push_back( *it ); }
    return v;
}
```

In this example, post-increment operations are performed on expensive iterators. Polyspace assumes that the C++ version is C++03. Use the analysis option `-cpp-version` by using the value `cpp03`.

In the function `deserialize`, the `for` loops iterate over a `std::istream_iterator< std::complex< double > >` iterator `it`. Because these iterators are expensive to copy, the post-increment operation is more expensive than the pre-increment operation. The expensive post-increment is unnecessary and inefficient because the incremented iterator is not used. Polyspace flags the post-increment operations.

#### Correction

You can fix these defects in several ways. For instance, replace the post-increment operations by pre-increment operations. If C++11 is available, you might want to use range-based for loops when you iterate over each element of a container.

```
#include <complex>
#include <iterator> // istream_iterator
#include <sstream> // istringstream
#include <string>
#include <vector>

std::vector< std::complex< double > > deserialize( const std::string& s )
{
    std::vector< std::complex< double > > v;
    std::stringstream iss( s );
    for( std::istream_iterator< std::complex< double > > it( iss );
         it != std::istream_iterator< std::complex< double > >(); ++it )
    { v.push_back( *it ); }
    return v;
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_POST_INCREMENT`
**Impact:** Low

## Version History

**Introduced in R2021b**

## See Also

`Find defects (-checkers)` | `C++ standard version (-cpp-version)` | `Compiler (-compiler)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive return by value

Functions return large output by value instead of by reference or pointer

## Description

This defect occurs when functions return large output objects by value instead of the object being returned by reference or pointer.

This checker is raised if both of these conditions are true:

- The address of the returned object remains valid after the `return` statement.
- The returned object is any of these:
  - A nontrivially copyable object. Returning such an object by value might require an external function call, which is more expensive than returning it by reference. To check whether an object is nontrivially copyable, use the function `std::is_trivially_copyable`. For more details about this function, see `std::is_trivially_copyable` in the C++ reference.
  - A large trivially copyable object. Returning a trivially copyable object by value is more expensive when the object is large.

This defect is not raised if the returned object is:

- Inexpensive to copy.
- A temporary object or a nonstatic local object.

### Risk

It is inefficient to return a large object by value when you can return the object by reference or pointer. Functions might inadvertently return a large object by value because of a missing & or *. Such inefficient return statements might not be noticed. Consider this code:

```
#include<string>
class Buffer{
public:
    //..
    const std::string getName() {
        return m_names;
    }
    //...
private:
    std::string m_names;
};
```

The class `Buffer` contains a large private object `m_names`. It is common to have a public getter function for such a private object, such as `getName`, which returns the large object `m_names`. Because the return type of `getNames` is set as `const std::string` instead of `const std::string&`, the function returns the large object by value instead of by reference. The expensive return by copy might not be noticed because this code compiles and functions correctly despite the missing &. For similar sources of inefficiency, see `Expensive pass by value` and `Expensive copy in a range-based for loop iteration`.

### Fix

To fix this defect, return objects by using references. When using C code, use pointers to avoid returning objects by value.

- To return objects from a C++ function by reference, set the return type of the function as a reference. For instance:

  ```
  #include<string>
  class Buffer{
  public:
      //..
      const std::string& getName() {
          return m_name;
      }
      //...
  private:
      std::string m_name;
  };
  ```

  The function `getName()` returns the large object `m_names` by reference because the return type of the function is `const std::string&`, which is a reference.
- Alternatively, use pointers to avoid returning objects by value. For instance, set the return type of `getName()` to `const std::string*`, and then return the address of `m_names` as `&m_names`.

  ```
  #include<string>
  class Buffer{
  public:
      //..
      const std::string* getName() {
          return &m_name;
      }
      //...
  private:
      std::string m_name;
  };
  ```

  By using a pointer, the function `getName()` avoids returning `m_names` by value. This method is useful in C code where references are not available.

  Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Expensive Return by Value in C++ Code

```
#include<string>
#include<memory>
#include<array>
#include<cstdint>
class Buffer{
private:
    static const size_t SIZE = 10;
    std::string m_name; //Nontrivially copyable type
    std::array<uint8_t,SIZE> m_byteArray; // Large trivially copyable type
```

```
    size_t m_currentSize; // Small trivially copyable
public:
    //...
    const std::string getName(){
        return m_name;
    }
    const std::array<uint8_t,SIZE> getByteArray(){
        return m_byteArray;
    }
    size_t getCurrentSize(){
        return m_currentSize;
    }
};
```

In this example, various private objects in the class `Buffer` are accessed by their getter functions.

- The large object `m_name` is returned by the getter functions `getName` by value. Returning this nontrivially copyable object by value is inefficient when it can be returned by reference instead. Polyspace flags the unction.
- The object `m_byteArray` is returned by the getter function `getByteArray` by value. Returning this large object by value is inefficient when it can be returned by reference instead. Polyspace flags the function.
- The function `getCurrentSize` returns the integer `m_currentSize` by value. Copying this small object is not inefficient. Polyspace does not flag the function.

**Correction**

To fix these defects, return large objects by using references as return types of the getter functions. For instance, set the return type of `getName` to `const std::string&` instead of `const std::string`.

```
#include<string>
#include<memory>
#include<array>
#include<cstdint>
class Buffer{
private:
    static const size_t SIZE = 10;
    std::string m_name; //Nontrivially copyable type
    std::array<uint8_t,SIZE> m_byteArray; // Large trivially copyable type
    size_t m_currentSize; // Small trivially copyable
public:
    //...
    const std::string& getName(){
        return m_name;
    }
    const std::array<uint8_t,SIZE>& getByteArray(){
        return m_byteArray;
    }
    size_t getCurrentSize(){
        return m_currentSize;
    }
};
```

### Expensive Return by Value in C Code

```
typedef struct _Circle{
    double Origin_abscissa;
    double Origin_ordinate;
    double Radius;
    char name[10];
}Circle;

const Circle getCircle(){
    static Circle SpecificCircle;
    //...
    return SpecificCircle;
}
```

In this example, the function `getCircle` returns the large `static` object `SpecificCircle` by value. Polyspace flags the function.

#### Correction

To fix this issue, return the object `SpecificCircle` by using a pointer. Declare the return type of the function `getCircle` as `const Circle*` instead of `const Circle`, and then return the address of the object, that is, &`SpecificCircle`.

```
typedef struct _Circle{
    double O_abscissa;
    double O_ordinate;
    double Radius;
    char name[10];
}Circle;

const Circle* getCircle(){
    static Circle SpecificCircle;
    //...
    return &SpecificCircle;
}
```

## Result Information

**Group:** Performance
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_RETURN_BY_VALUE`
**Impact:** Medium

# Version History

**Introduced in R2020b**

## See Also

`Find defects (-checkers)` | `Expensive pass by value` | `Expensive copy in a range-based for loop iteration` | `Expensive local variable copy`

**Topics**

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Expensive return of a const object

The return statement of a function copies an objects instead of moving it because the returned object is declared as a const

## Description

This defect occurs when these conditions are true:

- A function returns a const object, forcing a copy construction of the returned object from a const object.
- The returned object implements an available move constructor.

For instance, consider this code:

```
std::string foo(const std::string& str){
    const std::string cstr{str};
    //...
    return cstr;
}
```

The string cstr is constructed from a const string. Because cstr is a const, the return statement copies it instead of moving it. Polyspace flags the declaration of cstr.

### Risk

When returning objects, compilers implicitly attempt moving the returned object instead of copying it when the object supports move semantics. When you declare an object as a const, this implicit move is suppressed and the compiler copies the object. When the return statement copies the constructed return object instead of moving it, the code becomes less efficient. Because the inefficient code compiles and behaves correctly, the inefficiency might be difficult to diagnose.

### Fix

To fix this defect, make the returned object into a nonconst.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Avoid Returning const Objects

```
#include <string>

class Task
{
public:
    std::string exchangeName( const std::string& name )
    {
        const std::string old_name = m_name;
        m_name = name;
        return old_name;
    }
     //...
private:
    std::string m_name;
};
```

In this example, the function `exchangeName` returns the string `old_name`, which is a `const` object. The compiler cannot move `old_name`. The return statement of the function `exchangeName` is forced to copy the constructed return object, which is inefficient. Polyspace flags the declaration of `old_name`.

**Correction — Return Nonconst Objects**

To facilitate implicit move operation when returning an object, declare the returned objects as nonconst.

```
#include <string>

class Task
{
public:
    std::string exchangeName( const std::string& name )
    {
        std::string old_name = m_name;
        m_name = name;
        return old_name;
    }
     //...
private:
    std::string m_name;
};
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_RETURN_CONST_OBJECT`
**Impact:** Low

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"

“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Expensive use of a standard algorithm when a more efficient method exists

Functions from the `algorithm` library are misused with inappropriate inputs, resulting in inefficient code

## Description

Polyspace reports a defect when you misuse functions in the `algorithm` library with a container, which results in inefficient code. You might be recalculating known or constant information about the container or using a function that is inappropriate for the container. The scenarios that trigger this defect include:

- Running a search by calling functions such as `std::find`, `std::equal_range`, `std::upper_bound`, `std::binary_search`, and `std::lower_bound` with associative containers such as `std::set`, `std::map`, `std::multiset`, and `std::mutimap`.
- Checking for the presence of a certain key in a container by calling `std::count`. That is, you convert the output of `std::count` to a `bool` or compare the output to either `0` or `1`.
- Checking for a sorted associative container by calling `std::is_sorted`.
- Calculating the size of an associative container by calling `std::distance`.
- Searching for consecutive equal elements in unique-value containers such as `std::set` or `std::map` by calling `std::adjacent_find`.
- Passing expensive functors by value to standard functions.

### Risk

Misusing the functions from the `algorithm` library might result in inefficient code or unexpected behavior.

- Searching associative containers by using linear search functions, such as `std::find` and `std::binary_search`, is inefficient.
- When checking for a certain key in a container, performing an exhaustive search for every instance of the key by calling `std::count` is unnecessary and inefficient.
- If an associative container is already sorted, calling `std::is_sorted` is unnecessary. If the associative container is not sorted, the output of `std::is_sorted` might be indeterminate, resulting in unexpected behavior.
- Calculating the size of an associative container by calling its `size` method is more efficient than using the `std::distance` function.
- Calling `std::adjacent_find` with unique-value containers such as `std::set` or `std::map` always results in `end()`. Because the output is constant, the call is unnecessary.
- If you pass a functor to a standard algorithm function, it is always copied. If the functor is expensive, this copy operation makes your code expensive and inefficient.

### Fix

To fix this defect, refactor your code to use the standard `algorithm` functions more efficiently.

- When searching in associative containers, avoid using `std` functions. If possible, use the search method of the container, such as `std::set::find`.
- When checking for the presence of a certain key, call the `find` or `contains` (C++20) method of the container. For containers that do not implement these methods, use `std::find` instead of `std::count`.
- Avoid using `std::is_sorted` or `std::adjacent_find` with associative containers.
- To check the size of an associative container, use the `size` method of the container. Avoid using `std::distance` to calculate size.
- Avoid passing expensive-to-copy functors to `std` functions. Instead, create a wrapper functor that contains a reference to the original functor and pass the wrapper to standard algorithm functions.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Use of `std` Functions with Associative Containers

In this example, Polyspace flags the use of `std` functions, such as `std::find` or `std::count`, with associative containers when the containers have equivalent `find` or `count` methods. Using these `std` functions with containers that do not have equivalent methods is not reported as a defect. For instance,

- Polyspace flags the use of `std::find` with a `set` because `std::set::find` is more efficient.
- Polyspace does not flag the use of `std::find` with a `vector` because the class `vector` has no `find` method.

```
#include <unordered_set>
#include <unordered_map>
#include <forward_list>
#include <array>
#include <set>
#include <map>
#include <vector>
#include <deque>
#include <list>
#include <algorithm>
#include <string>

typedef std::string V;
typedef std::string K;

void lookup()
{
    std::string key;
    std::pair<const K, V> kv;


    std::set<V> s;
    std::multiset<V> ms;
    std::map<K, V> m;
    std::multimap<K, V> mm;
    std::vector<V> v;
    std::list<V> l;
    std::deque<V> d;
    std::string str;
    std::unordered_map<K, V> um;


    std::find(s.begin(), s.end(), key);
    std::equal_range(m.begin(), m.end(), kv);
    std::lower_bound(mm.begin(), mm.end(), kv);
    std::upper_bound(ms.begin(), ms.end(), key);
    std::count(um.begin(), um.end(), kv);
    if(std::binary_search(s.begin(), s.end(), key)==0){
          //....
      }

    std::find(v.begin(),v.end(), key); //Compliant
    std::equal_range(l.begin(),l.end(), key); //Compliant
    std::lower_bound(d.begin(),d.end(), key); //Compliant


}
```

**Correction — Use the Methods of Containers**

To fix this defect, use the methods of a container instead of the `std` functions. For instance, when using a `set`, use `std::set::find` instead of `std::binary_search`. It might be necessary to refactor your code to use the lookup methods of the container.

```
#include <unordered_set>
#include <unordered_map>
#include <forward_list>
#include <array>
#include <set>
#include <map>
#include <vector>
#include <deque>
#include <list>
#include <algorithm>
#include <string>

typedef std::string V;
typedef std::string K;


void lookup()
{
    std::string key;
    std::pair<const K, V> kv;

    std::unordered_map<K, V> um;
    std::set<V> s;
    std::multiset<V> ms;
    std::map<K, V> m;
    std::multimap<K, V> mm;
    std::vector<V> v;
    std::list<V> l;
    std::deque<V> d;
    std::string str;


    s.find(key);
    m.equal_range(key);
    mm.lower_bound(key);
    ms.upper_bound(key);
      if(s.find(key)==s.end()){
          //....
      }
      um.count(key);//Compliant


}
```

### Use of `std::count` to Check Containment

In this example, Polyspace flags the use of `std::count` to check if a container contains a particular member. Using the `find` method of the container or the `std::find` function is more efficient when checking containment.

Using `std::count` to count instances of a member does not cause a defect if the container does not have an equivalent member. For instance, Polyspace does not report a defect when you use `std::count` to check if `v` contains two instances of the string `key`.

```
#include <unordered_set>
#include <unordered_map>
#include <forward_list>
#include <array>
#include <set>
#include <map>
#include <vector>
#include <deque>
#include <list>
#include <algorithm>
#include <string>

typedef std::string V;
typedef std::string K;


void contains()
{
    std::string key;
    std::pair<const K, V> kv;

    std::set<V> s;
    std::multiset<V> ms;
    std::vector<V> v;
    std::unordered_map<K, V> um;

    //Check containment
    bool b_v = std::count(v.begin(), v.end(), key);
    bool b_um = std::count(um.begin(), um.end(), kv);
    bool b_s = std::count(s.begin(), s.end(), key);
    bool b_ms = std::count(ms.begin(), ms.end(), key);
    if(std::count(v.begin(), v.end(), key)==2){ //Compliant
        //...
    }

}
```

#### Correction — Use `find` to Check Containment

To fix these defects, use the `find` method of a container, such as `std::set::find`, to check for containment. When checking containment in a container that does not have a `find` method, use `std::find`.

```
#include <unordered_set>
#include <unordered_map>
#include <forward_list>
#include <array>
#include <set>
#include <map>
#include <vector>
#include <deque>
#include <list>
#include <algorithm>
#include <string>

typedef std::string V;
typedef std::string K;


void contains()
{
    std::string key;
    std::pair<const K, V> kv;

    std::set<V> s;
    std::multiset<V> ms;
    std::vector<V> v;
    std::unordered_map<K, V> um;

    //Check containment
    bool b_v = (std::find(v.begin(), v.end(), key)!=v.end());
    bool b_um = (um.find(key)!=um.end());
    bool b_s = (s.find(key)!=s.end());
    bool b_ms = (ms.find(key)!=ms.end());

}
```

#### Inappropriate Use of `std` Functions with Containers

In this example, Polyspace flags inappropriate uses of `algorithm` functions with container.

- The function `std::is_sorted` returns a known constant when you call the function with a `std::set` object. The function returns an indeterminate value when you call it with an `std::unoerderd_set` object. Because the function call either returns a constant or an indeterminate value, calling `std::is_sorted` on `std::set` or `std::unoerderd_set` is unnecessary. Polyspace flags these usages.

- An `std::set` or `std::map` object cannot contain duplicate elements by definition. Calling `std::adjacent_find` on these types of containers is unnecessary because the call always returns `end()`. Polyspace flags such usage.
- Associative containers have their own `size` methods. Polyspace flags the use of `std::distance` to measure the size of an associative container.

```
#include <unordered_set>
#include <unordered_map>
#include <forward_list>
#include <array>
#include <set>
#include <map>
#include <vector>
#include <deque>
#include <list>
#include <algorithm>
#include <string>

typedef std::string V;
typedef std::string K;

void contains()
{
    std::string key;
    std::pair<const K, V> kv;
    std::map<K, V> m;
    std::set<V> s;
    std::multiset<V> ms;
    std::vector<V> v;
    std::unordered_set< V> us;

    // Expression always returns same value
    bool b1 = std::is_sorted(s.begin(), s.end());

    // Result might be indeterminate
    bool b2 = std::is_sorted(us.begin(), us.end());

    // std::set never has duplicates
    std::adjacent_find(s.begin(), s.end());

    // std::map never has duplicates
    std::adjacent_find(m.begin(), m.end());

    //Inefficient use
    std::distance(s.begin(), s.end());
}
```

### Passing Expensive Functors to `std` Functions

In this example, you pass the functor `ExpensiveLess` to `std::stable_sort`. This function copies the expensive functor, resulting in inefficient code. Polyspace reports a defect.

```
#include <string>
#include <vector>
#include <algorithm>
#include <functional>
```

```
class ExpensiveLess {
public:
    ExpensiveLess() {}

    bool operator()(const std::string &s1, const std::string &s2) const
    {
        return s1 < s2;
    }

    std::vector<std::string> m_list;
};

void foo()
{
    ExpensiveLess f;
    std::vector<std::string> v, v1;

    std::stable_sort(v.begin(), v.end(), f); //Defect
}
```

**Correction — Use `std::reference_wrapper` Object**

To fix this defect, create a `std::reference_wrapper` object `wrapper`. This object contains a reference to an `ExpensiveLess` object. Passing this inexpensive wrapper might improve the efficiency of your code.

```
#include <string>
#include <vector>
#include <algorithm>
#include <functional>

class ExpensiveLess {
public:
    ExpensiveLess() {}

    bool operator()(const std::string &s1, const std::string &s2) const
    {
        return s1 < s2;
    }

    std::vector<std::string> m_list;
};

void foo()
{
    ExpensiveLess f;
    std::reference_wrapper<ExpensiveLess> wrapper(f);
    std::vector<std::string> v, v1;
    std::stable_sort(v.begin(), v.end(), wrapper); //No Defect
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off

**Command-Line Syntax:** `EXPENSIVE_USE_OF_STD_ALGORITHM`
**Impact:** Medium

## Version History

**Introduced in R2021b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive use of container's count method

The function member `count()` of a container is used for checking if a key is present, leading to inefficient code

## Description

This defect is raised when the function member `count()` of these containers is called for checking if a key is present:

- `std::multimap`
- `std::multiset`
- `std::unordered_multiset`
- `std::unordered_multimap`

When checking containment, you convert the output of the container's `count()` method to a `bool`, or compare it to either `0` or `1`.

### Risk

The `count` function of the preceding containers performs a linear search and finds all the instances of a key in the container. When checking if a key is present in a container, performing an exhaustive search for every instance of the key is unnecessary and inefficient.

### Fix

To fix this defect, call the member `find` or `contains` function of a container when checking for containment. These functions stop searching for a key as soon as one instance of the key is found. These functions check containment more efficiently compared to the `count` function.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Inefficient Containment Check

```
#include <unordered_set>
#include <unordered_map>
#include <set>
#include <map>
#include <algorithm>
#include <string>

typedef std::string V;
typedef std::string K;


void conatins()
{
    std::string key;
    std::pair<const K, V> kv;


    std::multiset<V> ms;
    std::multimap<K, V> mm;
    std::unordered_multimap<K, V> umm;
    std::unordered_multiset<K> ums;

    //Check containment

    bool b_ms = (ms.count(key)==0);
    bool b_mm = (mm.count(key)==0);
    bool b_ums = (ums.count(key)==0);
    bool b_umm = (umm.count(key)==0);

}
```

In this example, Polyspace flags the use of the member count function to check containment in various containers. These count functions are inefficient because they continue searching for `key` even after the first instance is found.

**Correction**

To fix this defect, use the member find function to check for containment. For instance, use `std::multimap::find` instead of `std::multimap::count`. The find functions are more efficient because they stop searching as soon as the first instance of `key` is found.

```
#include <unordered_set>
#include <unordered_map>
#include <set>
#include <map>
#include <algorithm>
#include <string>

typedef std::string V;
typedef std::string K;

void conatins()
{
    std::string key;
    std::pair<const K, V> kv;


    std::multiset<V> ms;
    std::multimap<K, V> mm;
    std::unordered_multimap<K, V> umm;
    std::unordered_multiset<K> ums;

    //Check containment

    bool b_ms = (ms.find(key)== ms.end());
    bool b_mm = (mm.find(key)== mm.end());
    bool b_ums = (ums.find(key)==ums.end());
    bool b_umm = (umm.find(key)==umm.end());

}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_CONTAINER_COUNT`
**Impact:** Medium

# Version History

**Introduced in R2021b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive use of container's insertion method

One of the insertion methods of a container is used to insert a temporary object

## Description

This defect occurs when you use a container's insertion method to insert a temporary object into a container. For instance:

```
std::vector<std::string> v;
v.push_back("foo");//Defect
```

This checker flags use of these methods to insert a temporary object:

- `insert()`
- `push()`
- `push_front()`
- `push_back()`

Most containers from the C++ standard template library implements these insertion methods and are supported by this checker.

### Risk

When you insert a temporary object by using a container's insertion method, the compiler constructs a temporary object, and then moves or copies the object into the container. Because this insertion method requires a move or copy operation in addition to a construction, it is less efficient. Using this method results in inefficient code.

### Fix

Avoid the unnecessary move or copy operation by constructing the object-to-be-inserted in place in the container by using the container's `emplace` methods. Such methods might include `emplace()`, `emplace_front()`, or `emplace_back()`. Because these methods require no copy or move operations, they are more efficient.

When using `emplace` methods, avoid supplying constructor functions manually. It is more efficient to allow the `emplace` methods to construct the object. For instance:

```
std::vector<std::string> v;
v.emplace_back(std::string("foo"));//Compliant but less efficient
v.emplace_back("foo");// Efficient fix
 std::map<int, std::string> m;
m.emplace(std::make_pair(5, "foo"));//Compliant but less efficient
m.emplace(5, "foo");// Efficient fix
```

Using `v.emplace_back(std::string("foo"))` or `m.emplace(std::make_pair(5, "foo"))` does not raise a violation of this checker. But it is more efficient to use `emplace` methods without specifying constructor functions, such as `m.emplace(5, "foo")`.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Avoid Inserting Temporary Objects By Using Container's Insertion Method

```
#include <vector>
#include <string>

void addNameToList(std::vector<std::string> &names, const char *pszName)
{
    names.push_back(pszName); //Defect
}
```

In this example, the method `push_back` is used to insert a temporary string into a vector. Using the `push_back` method necessitates constructing a string object, and then moving the object into the vector. The move operation results in inefficient code. Polyspace flags the method `push_back`.

#### Correction — Construct Object In Place

To resolve this defect, construct the object to be inserted in place by using `emplace_back`. This method avoids the unnecessary move operation, resulting in more efficient code.

```
#include <vector>
#include <string>

void addNameToList(std::vector<std::string> &names, const char *pszName)
{
    names.emplace_back(pszName);
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_CONTAINER_INSERTION`
**Impact:** Medium

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Expensive use of non-member `std::string operator+()` instead of a simple append

The non-member `std::string operator+()` function is called when the append (or +=) method would have been more efficient

## Description

This defect occurs when you append to a string using the non-member function `std::string operator+()`, for instance:

```
std::string s1;
s1 = s1 + "Other";
```

### Risk

The operation:

```
s1 = s1 + "Other";
```

invokes the non-member function `std::string operator+()` for the string concatenation on the right-hand side of the assignment. The function returns a temporary string, which is then assigned to `s1`.

Directly calling the member function `operator+=()` avoids the creation of this temporary string and is more efficient.

### Fix

To append to a string, use the member function `operator+=()`, for instance:

```
std::string s1;
s1 += "Other";
```

or the member function `append`, for instance:

```
std::string s1;
s1.append("Other");
```

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Appending to String Using `std::string operator+()`

```
#include <string>

void addJunior(std::string &name) {
    name = name + ", Jr.";
}
```

```
void addSenior(std::string &name) {
    name += ", Sr.";
}

void addDoctor(std::string &name) {
    name.append(", MD");
}
```

In this example, the checker flags the string append in the `addJunior` function, but not the ones in the other two functions. If the function `addJunior` is called several times in a loop, creation of a temporary string each time can be a significant performance issue.

### Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_STD_STRING_APPEND`
**Impact:** Low

## Version History

**Introduced in R2020b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive use of `std::string` method instead of more efficient overload

An `std::string` method is called with a string literal of known length, instead of a single quoted character

## Description

This defect occurs when you invoke certain `std::string` methods with a string literal of known length instead of a single quoted character. When certain `std::string` methods are called with a string literal, the method must compute the length of the literal even though the information is known at compile time. Polyspace flags such calls as inefficient. For instance, Polyspace flags the first two calls to `std::string::find`:

```
std::string str;
//...
str.find("A");//Inefficient
//...
str.find("ABC",offset,1);//Inefficient
str.find('A');//Efficient
```

In the first two calls, the compiler computes the length of the string literal "A" when it is already known before runtime. In the third call, the compiler does not compute the length of the input, making the call more efficient than the first two. Polyspace raises this defect when these `std::string` methods are invoked by using a string literal of known length instead of a single quoted character:

- `find`
- `rfind`
- `find_first_of`
- `find_last_of`
- `find_first_not_of`
- `find_last_not_of`
- `replace`
- `operator=`
- `operator+=`
- `starts_with` (C++20)
- `ends_with` (C++20)

### Risk

In some cases, you can call `std::string` methods with either a string literal or a single quoted character. In these cases, it is inefficient to call the `std::string` methods with string literal because such calls force the compiler to compute the lengths of the string literals, which is already known before runtime. Because `std::string` methods are frequently used, inefficient calls to these methods might result in expensive and inefficient code.

#### Fix

To fix this issue, call the `std::string` methods by using single quoted characters instead of a string literal when appropriate. For instance, you might use a single quoted character as an input instead of a string literal consisting of a single character or a repetition of a single character. You might need to use a different overload of the method that accepts a single quoted character.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Use Single Quoted Characters to Invoke `std::string` Methods

```
#include <string>
#include <set>
constexpr size_t ONE(){ return 1; }
void foo(std::string& str){
    int pos, count;
    str += "A";
    str += "\0";
    str.find("AA", 0, ONE());
    str.find("A");
    str.find_first_of("A");
    str.find_last_of("A");
    str.find_first_not_of("A");
    str.find_last_not_of("A");
    str.replace(0, 1, "A");
    str.rfind("AA", 0,1);
    str.replace(0, count, "AAAAA");
    str.replace(pos, count, "AB", 1);
}
```

In this example, several methods and operators of the `std::string` class are invoked to perform string operations. For these calls, you already know the length of the string literals before program runtime. Because the input is passed as a string object, the compiler performs a linear search to find its length, which is redundant. Polyspace flags such inefficient invocation of string operations.

#### Correction – Use Overloads that Accepts `char` as Input

To fix this issue, use a single quoted character instead of a string literal. For methods such as `replace`, rewrite the function call and use the overload that accepts `char` type variables when the input is a single character.

```
#include <string>
#include <set>
constexpr size_t ONE(){ return 1; }
void foo(std::string& str){
    int pos, count;
    str += 'A';
    str += '\0';
    str.find('A');
    str.find_first_of('A');
    str.find_last_of('A');
    str.find_first_not_of('A');
    str.find_last_not_of('A');
    str.replace(0, 1, 1,'A');//Rewritten call that accepts a single char
    str.rfind('A');
    str.replace(0, count, 5,'A');//Rewritten call that accepts a single char
    str.replace(pos, count, 1,'A');
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** EXPENSIVE_USE_OF_STD_STRING_METHODS
**Impact:** Low

# Version History

**Introduced in R2021a**

## See Also

Find defects (-checkers)

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive use of `std::string` with empty string literal

Use of `std::string` with empty string literal can be replaced by less expensive calls to `std::basic_string` member functions

## Description

In your C/C++ code, the checker flags these operations:

- Constructing an instance of `std::string` by using an empty string literal
- Assigning an empty string literal to an instance of `std::string`
- Comparing an instance of `std::string` to an empty string literal

Usage notes and limitations:

- The checker does not track the origin of `const char` pointer variables that are empty and are eventually used with `std::string`.
- This checker is partially obsolete when you use current compilers. Compilers such as GCC 5.1 and Visual Studio 2015 optimize out construction from an empty string literal and treat it as identical to the default construction.

### Risk

The preceding operations can be replaced by calls to the default constructor and the `empty` and `clear` member functions of the `std::basic_string` class template. Certain compilers might generate additional instructions for the explicit operations compared to the use of the built-in member functions. The use of these operations might reduce the performance of the compiled code.

### Fix

Replace explicit operations involving empty string literals by these calls to the default constructor and member functions of `std::basic_string`.

| Do not Use | Use |
|---|---|
| `std::string s("");` | `std::string s;` |
| `std::string s = "";` | `std::string s;` |
| `s = "";` | `s.clear();` |
| `if (s == "")` | `if (s.empty())` |
| `return "";` | `return {};` |
| `void foo(const std::string& s = "");` | `void foo(const std::string& s = {});`(C++11) or `foo(const std::string &str2 = std::string())` |
| `foo("");` | `foo({})` (C++11) or `foo(std::string())` |
| `Class::Class() : str("") {//...}` | `Class::Class() : str() {//...}` |

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Comparison With Empty String Literal

```
#include <iostream>
#include <string>

void compareString(const std::string &str1, const std::string &str2="")//Noncompliant
{
    if (str1 == "")//Noncompliant
    {
        std::cout << "The string is empty" << std::endl;
    }
    else
    {
        if (str1.compare(str2) != 0)
        std::cout << str1 << " is not " << str2 << '\n';
    }
}


void bar(){
    compareString("String1");
    compareString("String1","");//Noncompliant
}
```

In this example, three string operations are performed by using empty string literals. Polyspace flags these operations as inefficient.

#### Correction — Use Member Functions of `std::string`

To fix the flagged issues, replace string operations with empty string literals with calls to member functions of `std::string`. For instance:

- Replace `const std::string &str2=""` with `const std::string &str2={}`
- Replace `if(str1 == "")` with `if(str1.empty())`
- Replace `compareString("String1","")` with `compareString("String1",{})`

```
#include <iostream>
#include <string>

void compareString(const std::string &str1, const std::string &str2 = {})//Compliant
{
    if (str1.empty())//Cmpliant
    {
        std::cout << "The string is empty" << std::endl;
    }
    else
    {
        if (str1.compare(str2) != 0)
        std::cout << str1 << " is not " << str2 << '\n';
    }
}


void bar(){
    compareString("String1");
    compareString("String1",{});//Compliant
}
```

## Result Information

**Group:** Performance<br>
**Language:** C++<br>
**Default:** Off<br>
**Command-Line Syntax:** `UNNECESSARY_EMPTY_STRING_LITERAL`<br>
**Impact:** Low

# Version History

**Introduced in R2021a**

## See Also

`Find defects (-checkers)` | `Expensive constant std::string construction` | `Expensive std::string::c_str() use in a std::string operation`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive use of string functions from the C standard library

String functions from the C standard library are used inefficiently

## Description

This defect occurs when any of these conditions are true:

- You use `strcmp()` or `strlen()` to check if a C-string is empty.
- You use `strcpy()` to clear an existing C- string.

### Risk

When determining if a C-string is empty, it is not efficient to compare its length to zero by using `strlen()` or compare the C-string to an empty string (`""`) by using `strcmp()`. The function `strlen()` checks the entire C-string until it finds a null, which is inefficient when checking if a C-string is empty. Invoking these functions to check for empty C-strings is inefficient and might result in unnecessary function call overhead.

Similarly, using `strcpy()` to clear a C-string is unnecessary.

### Fix

To fix this defect:

- When checking if a C-string is empty, avoid using `strcmp()` or `strlen()`. Instead, compare the first character of the C-string to `'\0'`.
- When clearing a C-string, avoid using `strcpy()`. Instead, assign the `'\0'` character to the C-string.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Avoid Using `strcmp()` or `strlen()` to Check for Empty C-String

```
#include <cstring>


int isEmpty(const char* pszName)
{
   return strcmp(pszName, "") == 0; //Defect
}

int isEmpty2(const char* pszName)
{
   return strlen(pszName) == 0;  //Defect
}
```

In this example, Polyspace flags the use of string functions `strcmp()` and `strlen()` to check for empty C- strings. Checking for empty C-strings is not the intended use of these functions. Such use of these functions results in inefficient and unnecessarily complex code.

#### Correction — Check for Empty C-String by Comparing First Character to `'\0'`

Instead of using the string functions, it is more efficient to check for empty C-strings by comparing the first character of the string to null or `'\0'`. This method reduces the function overhead and makes the code more efficient.

```
#include <cstring>


int isEmpty(const char* pszName)
{
   return *pszName == 0;
}
```

### Avoid Clearing C-Strings by Using `strcpy()`

```
#include <string.h>

void foo(void){
    char* str;
    //...
    strcpy(str,"");
    //...
}
```

In this example, a C-string is cleared by copying an empty string into it. Calling `strcpy` to clear a C-string is unnecessary and makes the code inefficient.

#### Correction

To clear a C-string, assign null or `'\0'` to the string.

```
#include <string.h>

void foo(void){
    char* str;
    //...
    *str = '\0';
    //...
}
```

## Result Information

**Group:** Performance  
**Language:**C | C++  
**Default:** Off  
**Command-Line Syntax:** `EXPENSIVE_USE_OF_C_STRING_API`  
**Impact:** Low

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive use of substr() to shorten a std::string

The method std::string::substr() is called to shorten an std::string object

## Description

This defect occurs when you call std::string::substr() to shorten an std::string up to the n^th character:

```
std::string str;
str = str.substr(0, n); ;//Defect
```

### Risk

When you call std::string::substr() to shorten or truncate a string, the compiler first constructs a temporary string containing the smaller substring, and then assigns the temporary string object to the original string. Shortening a string by calling std::string::substr() requires constructing a temporary string, which is unnecessary. This method results in inefficient code.

### Fix

Instead of calling std::string::substr(), use std::string::resize() to shorten a string. When you shorten strings by using the method std::string::resize(), the end of the string is moved to the location that you want without requiring any additional construction. The method std::string::resize() is more efficient than std::string::substr() for shortening strings.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Avoid Using std::string::substr() to Shorten Strings

```
#include<string>
void shortenString(std::string& str, size_t pos){
    str = str.substr(0, pos);//Defect
}
```

In this example, the substr() method is called to shorten the string str. This method creates a temporary string object containing the first pos characters of str, and then overwrites str with the temporary string. This method of shortening a string object is inefficient and Polyspace flags it.

#### Correction — Use std::string::resize()

To resolve this defect, use the resize() method to shorten the string. This method does not create any new string objects and is more efficient.

```
#include<string>
void shortenString(std::string& str, size_t pos)
{
    if (pos < str.size()) {
        str.resize(pos);
    }
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_STD_STRING_RESIZE`
**Impact:** Low

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Inefficient string length computation

String length calculated by using string length functions on return from `std::basic_string::c_str()` instead of using `std::basic_string::length()`

## Description

This defect occurs when the length of a `std::basic_string` string is calculated by using string length functions on the pointer returned from `std::basic_string::c_str()` instead of using the method `std::basic_string::length()`.

The checker flags string length functions such as `strlen`, `wcslen` and `char_traits::length`.

### Risk

`std::basic_string::c_str()` returns a pointer to a null-terminated character array that stores the same data as the data stored in the string. Using a string length function such as `strlen` on this character array is expected to return the string length. This approach might seem superficially equivalent to using the `std::basic_string::length()` method for the string length.

However, the function `strlen(str)` is of linear complexity O(N) where N is the length of string `str`. If `str` is of type `std::basic_string`, this complexity is unnecessary since calling the `std::basic_string::length()` method returns the length more efficiently (with complexity O(1)).

### Fix

If a string is of type `std::basic_string`, to get the string length, instead of using string length functions such as `strlen`, for instance:

```
std::string s;
auto len = strlen(s.ctr());
```

use the `std::basic_string::length()` method, for instance:

```
std::string s;
auto len = s.length();
```

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Result Information

**Group:** Performance<br>
**Language:** C++<br>
**Default:** Off<br>
**Command-Line Syntax:** `INEFFICIENT_BASIC_STRING_LENGTH`<br>
**Impact:** Medium

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Inefficient use of for loop

Range-based for loop can perform equivalent iteration more efficiently

## Description

This defect occurs when you use `for` loops that loop through all elements of a container or a C-style array and both of these conditions are true:

- The `for` loop has only one loop counter variable. For instance, Polyspace does not flag this `for` loop:

  ```
  for(int i=0, j=0; i < 10; ++i, ++j){
  //...
  }
  ```
- The `for` loop uses the loop counter variable to access only the elements of the container or the C-style array. For instance, in this code snippet, Polyspace flags the first loop but not the second one.

  ```
  #include <iostream>

  int myArray[10]

  void cArray()
  {
      //First loop: Loop counter used only to access elements of myArray
      for (int i = 0; i < 10; ++i) {
          myArray[i] = 0;
      }

      //Second loop: Loop counter assigned to elements of myArray
      for (int i = 0; i < 10; ++i) {
          myArray[i] = i;
      }


  }
  ```

Polyspace does not raise an **Inefficient use of for loop** defect in these scenarios:

- You iterate through a user-defined container. For instance, this `for` loop is compliant.

  ```
  template<typename T>
  class customContainer {
    typedef T* iterator;
    public:
      iterator begin();
      iterator end();
  };

  void func() {
    customContainer<int> myContainer;

    for(auto it = myContainer.begin(); it != myContainer.end(); it++) {
      std::cout << *it;
  ```

```
  }
}
```

Polyspace raises a defect only for C-style arrays and Standard Template Library containers.

- You use reverse iterators to loop through the elements of the C-style array or container. For instance, in this code snippet, there is no defect because you cannot use a range-based `for` loop to perform an equivalent operation.

```
#include <iostream>
#include <vector>
#include <cstdint>

std::vector<std::uint32_t> myVector(10);

void myContainer()
{
    //loop uses reverse iteration
    for (auto it = myVector.rbegin(); it != myVector.rend(); ++it) {

        std::cout << *it;
    }
}
```

- If you loop through arrays:

  - You loop through the elements of multiple arrays or you loop through the elements of a multidimensional array.
  - The array size is unknown.

For instance, Polyspace does not flag these `for` loops:

```
int myArray[10];
int myOtherArray[10];
int multiArray[10][10];

void cArray()
{
    //loop through multiple array
    for (int i = 0; i < 10; ++i) {
        myArray[i] = 0;
        myOtherArray[i] = 0;
    }
    //loop through 2-dimensional array
    for (int i = 0; i < 10; ++i) {
        multiArray[i][i] = 0;
    }

}

void unknownSize(int someArray[])
{
    //loop through array of unknown size
    for (int i = 0; i < 10; ++i) {
        someArray[i] = 0;
    }
}
```

#### Risk

If you loop through all the elements of a container or a C-style array and you use the loop counter variable to access only the value of the elements, a `for` loop is slower, less readable, and more difficult to maintain than an equivalent range-based `for` loop.

#### Fix

Replace the `for` loop with a range-based `for` loop, which makes your code run more efficiently, more readable, and easier to maintain.

## Examples

### Inefficient Use of a `for` Loop

```
#include <iostream>
#include <vector>
#include <cstdint>

std::vector<std::uint32_t> myVector(10);

void myContainer()
{
    for (auto it = myVector.begin(); it != myVector.end(); ++it) {
        std::cout << *it;
    }
}
```

In this example, Polyspace flags the `for` loop, which iterates through all the elements of vector `myVector` and uses the loop counter `it` to only print the value of each element. To run this example, specify C++ version as `cpp11`.

#### Correction — Use a Range-Based `for` Loop Instead

One possible correction is to use a range-based `for` loop to perform an equivalent operation. The range-based `for` loop uses less code and makes your code run more efficiently, more readable, and easier to maintain

```
#include <iostream>
#include <vector>
#include <cstdint>

std::vector<std::uint32_t> myVector(10);

void myContainer()
{
    for (auto it : myVector) {
        std::cout << it;
    }
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `PREFER_RANGE_BASED_FOR_LOOPS`
**Impact:** Low

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

# Inefficient use of sprintf

The function sprintf, snprintf, or swprintf copies strings instead of the more efficient strcpy, strncopy, or wcsncpy

## Description

This checker is triggered when you use sprintf, snprintf, or swprintf to copy strings.

### Risk

The functions sprintf, snprintf, or swprintf are complex function with a variable argument list. Before executing these functions, the compiler parses the argument list to determine the argument types, which adds overhead to the code. Handling the different input formats that these functions support makes the function difficult to optimize. For instance, even if you want to copy only strings by using sprintf, the function must still support copying integers. Using these functions for copying strings make your code inefficient.

### Fix

To fix this defect, refactor your code and use dedicated functions such as strcpy, strcat, strncopy, wcsncpy or their variants to copy strings.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Inefficient Copying of C-Style Strings

```
#include <string.h>
#include <stdio.h>
#include <stddef.h>

void foo(char* p1, const char* p2, const char* p3, size_t N){
    //...
    sprintf(p1, "%s", p2); //Defect
    sprintf(p1, "%s", "String");//Defect
    sprintf(p1, "NoFormatting"); //Defect
    sprintf(p1, ""); //Defect
    sprintf(p1, "%s%s", p2, p3);//Defect
    snprintf(p1, N, "%s", p2); //Defect
    snprintf(p1, N, p2);//Defect
}

void foo_w(wchar_t* w1, const wchar_t* w2, size_t N){
    swprintf(w1, N, L"%s", w2); //Defect
    swprintf(w1, N, w2);  //Defect
}
```

```
void bar(char* p1, const char* p2, const char* p3){
    sprintf(p1, "%d", 5);//No Defect
    sprintf(p1, "%s%d", "String", 123);//No Defect
    int n = sprintf(p1, "%s", p2);//No Defect
}
```

In this example, the C strings `p2` and `p3` are copied into `p1` by calling `sprintf` and `snprintf`. Polyspace flags these inefficient copying. In the function, `foo_w`, the wide string `w2` is copied into `w1` by calling `swprintf`. Polyspace flags the inefficient copying. In the function `bar`, `sprintf` is called for formatting strings. When you use `sprintf` for formatting strings or use the return value of the function, Polyspace does not flag the call as inefficient.

#### Correction

To fix this defect, use dedicated string copying functions such as `strcpy`, `strncopy`, or `wcsncpy`.

```
#include <string.h>
#include <stdio.h>
#include <stddef.h>

void foo(char* p1, const char* p2, const char* p3, size_t N){
    //...
    strcpy(p1, p2); //Compliant
    strcpy(p1, "String");//Compliant
    strcpy(p1, "NoFormatting"); //Compliant
    *p1 = '\0';//Compliant
    strcpy(p1, p2); //Compliant
    strcat(p1, p3);//Compliant
    strncpy(p1, p2, N);//Compliant

}

void foo_w(wchar_t* w1, const wchar_t* w2, size_t N){
    wcsncpy(w1, w2, N);//Compliant
}
```

## Result Information

**Group:** Performance
**Language:**C | C++
**Default:** Off
**Command-Line Syntax:** `INEFFICIENT_SPRINTF`
**Impact:** Medium

# Version History

**Introduced in R2021b**

## See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"

“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Missing constexpr specifier

constexpr specifier can be used on variable or function for compile-time evaluation

## Description

This defect identifies potential constexpr variables and functions. For instance, the defect occurs if:

- You omit the constexpr specifier when a variable is initialized by using an expression that can be evaluated at compile time.

  The defect checker flags a local variable definition without the constexpr specifier if the variable is initialized with one of the following and not modified subsequently in the code:

  - An expression involving compile-time constants only.
  - Calls to a function with compile-time constants as parameters, provided the function is itself constexpr or the function contains only a return statement involving its parameters.
  - A constructor call with a compile-time constant, provided all member functions of the class including the constructor are themselves constexpr.

  The checker does not flag local, static variables.
- You omit the constexpr specifier from functions with a single return statement that could potentially be evaluated at compile time (given compile-time constants as arguments).

### Risk

If a variable value is computed from an expression that involves compile-time constants only, using constexpr before the variable definition, like this:

```
constexpr double eValSquared = 2.718*2.718;
```

ensures that the expression is evaluated at compile time. The compile-time evaluation saves on run-time overheads. Sometimes, the performance gains at run time can be significant.

If the expression cannot be evaluated at compile time, the constexpr keyword ensures that you get a compilation error. You can then fix the underlying issue if possible.

Note that the const keyword does not guarantee compile-time evaluation. The const keyword simply forbids direct modification of the variable value after initialization. Depending on how the variable is initialized, the initialization can happen at compile time or run time.

### Fix

Add the constexpr specifier to the variable or function definition.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Potential constexpr Variables and Functions

```
double squareIfPositive(double val) {
    return val > 0? (val * val): 0;
}

void initialize(void) {
    double eVal = 2.718;
    double eValSquare = squareIfPositive(2.718);
}
```

In this example, the checker flags the definition of squareIfPositive because the function contains a single return statement involving only its parameter val. Therefore, if val is a compile-time constant, the function can be evaluated at compile-time and can be a constexpr function.

The checker also flags the definition of eValSquare because it is initialized with a potentially constexpr function that takes a compile-time constant as argument.

#### Correction – Add constexpr Specifier

Add constexpr specifiers to the variable and function definitions.

```
constexpr double squareIfPositive(double val) {
    return val > 0? (val * val): 0;
}

void initialize(void) {
    constexpr double eVal = 2.718;
    constexpr double eValSquare = squareIfPositive(2.718);
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** MISSING_CONSTEXPR
**Impact:** Medium

# Version History

**Introduced in R2020b**

## See Also

Find defects (-checkers)

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Move operation uses copy

A move constructor or move assignment operator uses copy operations on base classes or data members

## Description

This defect is raised if all of these conditions are true:

- A move constructor or a move assignment operator of a class copies a base class or data member.
- Move operations for the base class or data members are available.
- The base class or data members are more expensive to copy than to move.

For instance, in this code the move constructor copies the data member `data`. Polyspace flags the copy operation.

```
class wrapper{
    //...
    wrapper(wrapper&& rhs): data(rhs.data){
        //...
    }

    private:
    std::string data;
}
```

### Risk

Developers often assume that move operations are cheaper than copy operations. They use move operations when dealing with large resources that are expensive to copy. Inadvertently omitting calls to `std::move` might cause a move constructor or a move assignment operator to copy data members and base classes, making the code inefficient. Because such unexpected move operations compile and run correctly, the inefficiency inducing omission of `std::move` might be difficult to detect.

### Fix

To fix this defect, use `std::move` to move base classes and data members of a class. As a best practice, use the default implicit move constructor and move assignment operator by setting them as `=default`. Instead of managing raw resources, use smart containers to enable the default move operations to move the resources correctly.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Avoid Copying Base Classes and Data Members in Move Operations

```
#include <memory>
#include <string>
```

```
#include <utility>

class ManagerInterface;
// ...
class UtilInterface
{
public:
    // ...
    UtilInterface( UtilInterface&& other )
      : m_name( other.m_name ),
        m_manager( other.m_manager )
    {/**/}
    UtilInterface& operator=( UtilInterface&& other )
    {
        m_name = other.m_name;
        m_manager = other.m_manager;
        return *this;
    }
    // ...
private:
    // ...
    std::string m_name;
    std::shared_ptr< ManagerInterface > m_manager;
    // ...
};
```

In this example, the move operations of the class `UtilInterface` copy the data member instead of moving them by using `std::move`. Polyspace flags the unexpected copy operations.

**Correction**

To fix the defect, use `std::move` to move the data members.

```
#include <memory>
#include <string>
#include <utility>

class ManagerInterface;
// ...
class UtilInterface
{
public:
    // ...
    UtilInterface( UtilInterface&& other )
      : m_name( std::move(other.m_name) ),
        m_manager( std::move(other.m_manager) )
    {}
    UtilInterface& operator=( UtilInterface&& other )
    {
        m_name = std::move(other.m_name);
        m_manager = std::move(other.m_manager);
        return *this;
    }
    // ...
private:
    // ...
    std::string m_name;
    std::shared_ptr< ManagerInterface > m_manager;
    // ...
};
```

#### Correction

Another method of fixing this defect is to declare the move assignment operator and the move constructor as `=default`.

```
#include <memory>
#include <string>
#include <utility>

class ManagerInterface;
// ...
class UtilInterface
{
public:
    // ...
    UtilInterface( UtilInterface&& other ) = default;//Compliant
    UtilInterface& operator=( UtilInterface&& other ) = default;//Compliant
    // ...
private:
    // ...
    std::string m_name;
    std::shared_ptr< ManagerInterface > m_manager;
    // ...
};
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `MOVE_OPERATION_USES_COPY`
**Impact:** Medium

# Version History

**Introduced in R2021b**

## See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# `std::endl` may cause an unnecessary flush

`std::endl` is used instead of the more efficient `\n`

## Description

This defect flags uses of `std::endl` in I/O operations and allows you to use the more efficient alternative, `\n`.

### Risk

`std::endl` inserts a newline (`\n`) followed by a flush operation. For instance:

```
std::cout << "Some content" << std::endl;
```

is equivalent to:

```
std::cout << "Some content" << '\n' << std::flush;
```

The implicit flush operation might not be necessary or intended. If your program has many I/O operations that use `std::endl`, the implicit flush operation can significantly reduce program performance. Since the flush operation is implicit, in case of a performance issue, it will be difficult to track the root cause of the issue.

### Fix

Use `\n` to enter a newline wherever possible.

If you require a flush operation, instead of `std::endl`, use `\n` followed by an explicit flush operation, for instance:

```
std::cout << "Some content" << '\n' << std::flush;
```

In this case, the analysis considers your use of a flush operation as deliberate and does not flag the use.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Possible Performance Impact from `std::endl` Use

```
#include <fstream>
using namespace std;

int main()
{
  ofstream aFile("file.txt");
  for ( int i = 0; i < 100000; i++) {
    aFile << "Hello World " << std::endl ;
  }
  aFile.close();
  return 0;
}
```

In this example, an `std::endl` is used in a loop during a write operation on a file. Since the loop has 100000 iterations, the slight delay from each implicit flush operation can add up to a significant reduction of performance.

#### Use \n and Avoid Flush

In a loop with several iterations, avoid the performance reduction in I/O operations by using `\n` instead of `std::endl`.

```
#include <fstream>
using namespace std;

int main()
{
  ofstream aFile("file.txt");
  for ( int i = 0; i < 100000; i++) {
    aFile << "Hello World \n" ;
  }
  aFile.close();
  return 0;
}
```

### Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `STD_ENDL_USE`
**Impact:** Low

## Version History

**Introduced in R2020a**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# std::move called on an unmovable type

std::move used on a class type with no move constructor or move assignment operator

## Description

This defect occurs when you use std::move to move an object of a class type that does not have a move constructor or move assignment operator.

### Risk

The use of std::move in statements such as:

```
Obj objTo {std::move(objFrom)};
objTo = std::move(objFrom);
```

indicates that you want to benefit from the performance gains of a move operation. However, because of the missing move constructor or move assignment operator, a copy operation happens instead.

If the class is expensive to copy, the unintended copy operation can cause a loss of performance.

### Fix

To make an object of type T movable, add a move constructor:

```
T (T&&);
```

and move assignment operator:

```
T& operator=(T&&);
```

to the class T. If the class does not have to directly manage a resource, you can use compiler-generated move operators using the =default syntax, for instance:

```
T (T&&) = default;
```

Otherwise, if a move operation is not required, remove the std::move call and directly copy the object.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### std::move Used with Unmovable Types

```
#include <utility>
#include <string>

class stringPair {
    std::string str1;
    std::string str2;
    public:
        stringPair(const stringPair & aPair); //Copy constructor
        stringPair& operator=(const stringPair & aPair);  //Copy assignment operator
};

void exchangePairs (stringPair& first, stringPair& second) {
    stringPair tempPair {std::move(first)};
    first = std::move(second);
    second = std::move(tempPair);
}
```

In this example, the type `stringPair` does not have a move constructor or move assignment operator. Uses of `std::move` on objects of this type result in copy operations instead.

**Correction – Add Move Constructor and Move Assignment Operator**

Make the type `stringPair` movable by adding a move constructor or move assignment operator.

```
#include <utility>
#include <string>

class stringPair {
    std::string str1;
    std::string str2;

    public:
        stringPair(const stringPair & aPair); //Copy constructor
        stringPair(stringPair && aPair) noexcept; //Move constructor
        stringPair& operator=(const stringPair & aPair);  //Copy assignment operator
        stringPair& operator=(stringPair && aPair) noexcept; //Move assignment operator
};

void exchangePairs (stringPair& first, stringPair& second) {
    stringPair tempPair {std::move(first)};
    first = std::move(second);
    second = std::move(tempPair);
}
```

**Correction -- Remove `std::move` call**

If you do not want to make the type `stringPair` movable, omit the `std::move` calls.

```
#include <utility>
#include <string>

class stringPair {
    std::string str1;
    std::string str2;

    public:
        stringPair(const stringPair & aPair); //Copy constructor
        stringPair& operator=(const stringPair & aPair);  //Copy assignment operator
};

void exchangePairs (stringPair& first, stringPair& second) {
    stringPair tempPair {first};
    first = second;
    second = tempPair;
}
```

### Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `STD_MOVE_UNMOVABLE_TYPE`
**Impact:** Medium

## Version History

**Introduced in R2020b**

### See Also

`Find defects (-checkers)` | `Const std::move input may cause a more expensive object copy` | `Const rvalue reference parameter may cause unnecessary data copies`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unnecessary padding

Members of a `struct` are padded to fulfill alignment requirement when rearranging the members to fulfill this requirement saves memory

## Description

This checker flags a `struct` object where the arrangement of its members necessitates additional padding to fulfill alignment requirement. Rearranging the members of such a `struct` object might fulfill the alignment requirement without requiring any additional padding. Because the padding is unnecessary for alignment purposes, eliminating the padding saves memory. Consider this `struct` in a 64bit system:

```
struct A {
     uint32_t m1;// 4 bytes
     uint64_t m2;// 8 bytes
     uint32_t m3;// 4 bytes
 };
```

To maximize speed, C/C++ requires that a variable be read in one cycle if possible. In this system, 8 bytes can be read during one cycle. If `m1` and `m2` are placed consecutively, the machine requires two cycles to read `m2`. Instead, the variable `m1` is placed in a 8 byte slot by itself after padding it by 4 bytes. Then `m2` is placed in its own 8 byte slot. The variable `m3` is also padded to fulfill alignment requirement for the `struct A`. Because of the padding, the size of `A` is 24 bytes even though the combined size of `m1`, `m2`, and `m3` is 16.

Polyspace raises this defect when alignment requirement can be fulfilled by rearranging the members of a `struct`. For instance, rearranging the members of `A` can eliminate padding:

```
struct A {
     uint64_t m2;// 8 bytes
     uint32_t m1;// 4 bytes
     uint32_t m3;// 4 bytes
 };
```

Here, `m2` is placed first in a 8 byte slot. Then `m1` and `m3` are placed together in another 8 byte slot. This rearrangement eliminates the padding.

### Risk

Unnecessary padding wastes memory, which can have several adverse impacts:

- Using more memory than necessary might exhaust the available memory, resulting in paging fault.
- Functions such as `memcpy` and `memcmp` might take longer.

### Fix

To fix this defect, rearrange the members of the `struct` to eliminate the unnecessary padding. Declare the largest `struct` members first, and then keep the declarations of same-sized members together. You might also use `pragma` directives to eliminate padding.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Unnecessary Padding in C-Style `struct`

```
#include <stdint.h>
struct GlobalStructA {
    uint32_t m1;
    uint64_t m2;
    uint32_t m3;
};

// Using pragma pack to eliminate padding
//
#pragma pack(push, 1)
struct GlobalStructPragmaPack {
    uint8_t  m1;
    uint64_t m2;
    uint8_t  m3;
};
#pragma pack(pop)
struct Array
{
    uint8_t  m1[5];
    uint64_t m2;
    uint8_t  m3[3];
};
struct StructWithBitField
{
    uint8_t  m1 : 1;
    uint8_t     : 1;
    uint8_t     : 1;
    uint8_t  m2 : 1;

    uint64_t m3;

    uint8_t  m4 : 1;
    uint8_t     : 1;
    uint8_t     : 1;
    uint8_t  m5 : 1;
};
```

In this example, Polyspace flags C style `struct` objects that contain unnecessary padding. Polyspace assumes that the processor has a 32-bit word length by default. Use the option `-target x86_64` to run this example.

- Because the 32-bit variable `m1` is declared first in the `GlobalStructA` object, it is padded to 64 bit. Polyspace flags the object because the padding can be eliminated by declaring `m1` after `m2`.
- The object `GlobalStructPragmaPack` has a similar issue with the order of its member declaration, but the padding is eliminated by a `pragma` directive. Polyspace does not flag this object.

- In the object Array, because m1[5] is declared before m2, the 40-bit array is padded to 64 bit. The 24-bit array m3 is padded to 64 bit. These paddings can be eliminated by declaring m2 first. Because Array contains unnecessary padding, Polyspace flags the object.
- In the object StructWithBitField, there are two bit-fields, each consisting of 4 bits. Because m3 is declared between these two bitfields, they are padded. Polyspace flags the object.

#### Correction

To fix this defect, change the order of the declaration to eliminate padding. For instance:

```
#include <stdint.h>
struct GlobalStructA {
    uint64_t m2;
    uint32_t m1;
    uint32_t m3;
};

// Using pragma pack to eliminate padding
//
#pragma pack(push, 1)
struct GlobalStructPragmaPack {
    uint64_t m2;
    uint8_t  m1;
    uint8_t  m3;
};
#pragma pack(pop)
struct Array
{
    uint64_t m2;
    uint8_t  m1[5];
    uint8_t  m3[3];
};
struct StructWithBitField
{
    uint64_t m3;

    uint8_t  m1 : 1;
    uint8_t     : 1;
    uint8_t     : 1;
    uint8_t  m2 : 1;

    uint8_t  m4 : 1;
    uint8_t     : 1;
    uint8_t     : 1;
    uint8_t  m5 : 1;
};
```

### Unnecessary Padding in class

```
#include <string>
#include<stdint.h>
class GlobalClassWithString {
    uint32_t    m1;
    std::string m2;
    uint32_t    m3;
};
class GlobalClassC {
  public:
    uint32_t m1;
  protected:
    uint64_t m2;
  private:
    uint32_t m3;
};
```

Polyspace assumes that the processor has a 32-bit word length by default. Use the option `-target x86_64` to run this example. Because of the order in which the members of the preceding classes are declared, the classes contain unnecessary padding. Polyspace flags these classes. The `public`, `private`, or `protected` labels of the members do not impact how they are organized in the memory.

#### Correction

To fix this defect, change the order of the declaration to eliminate padding. For instance:

```
#include <string>
#include<stdint.h>
class GlobalClassWithString{
    std::string m2;
    uint32_t    m1;
    uint32_t    m3;
};
class GlobalClassC {
  protected:
    uint64_t m2;
  public:
    uint32_t m1;
  private:
    uint32_t m3;
};
```

### Unnecessary Padding in 16-Bit Processor

```
#include <stdint.h>
struct SmallPadding {     //Defect in 16-bit, no defect in 32-bit
    uint8_t m1;
    uint8_t m2;
    uint8_t m3;
    uint8_t m4;
    uint8_t m5;
    uint8_t m6;
    uint8_t m7;
    uint32_t m8;
    uint8_t m9;
    uint8_t m10;
    uint8_t m11;
    uint8_t m12;
    uint8_t m13;
    uint8_t m14;
    uint8_t m15;
};
```

This example shows the sensitivity of this checker to the processor word length. In the object `SmallPadding`, there are 16 bits of padding, which is small compared to the total size of the `struct`. When the processor word length is large, the small padding is unlikely to cause inefficiency. For instance, reading `SmallPadding` requires three cycles regardless of the padding if the processor has a 64-bit alignment. Polyspace does not flag the object in this case. When the processor word length is smaller, the padding can lead to inefficiency. For instance, if the processor alignment is 16 bit, the added padding might cause inefficiency in reading the object. Polyspace flags `SmallPadding` in this case. To set the alignment of processor to 16 bit in a Polyspace analysis, use the options `-target mcpu -align 16`. See Generic target options on page 2-45.

## Result Information

**Group:** Performance<br>
**Language:** C | C++<br>
**Default:** Off<br>
**Command-Line Syntax:** `UNNECESSARY_STRUCT_PADDING`<br>
**Impact:** Medium

# Version History

**Introduced in R2021b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Unnecessary use of std::string::c_str() or equivalent string methods

Instead of a std::string object, a string operation uses the C-string obtained from std::string functions including std::string::c_str, std::string::data(), std::string::at(), or std::string::operator[], resulting in inefficient code

## Description

This defect occurs when a string operation is performed by using a C-string pointer obtained from string functions such as std::string::c_str, std::string::data(), std::string::at(), and std::string::operator[]. For instance, this checker is raised when:

- A new std::string or std::wstring is implicitly or explicitly constructed from the C-string obtained from a string function. This situation arises when a function expecting a const reference to the string encounters a const char* instead.
- A new copy of a string object is created explicitly from the C-string obtained from a string function. Using the copy constructor is the more efficient way of copying the string object.
- Certain std::string member functions are invoked by using the C-string obtained from a string function. Flagged functions include replace, append, assign, compare, and find. Using an std::string object directly to invoke std::string member functions is more efficient.
- A user-defined function that is overloaded to accept either of the const char* or const std::string arguments is invoked by using a C-string pointer. It is more efficient to invoke the std::string overload of such a function. When a function is overloaded in this way, calling the const char* overload from the body of the const std::string overload by using the C-string pointer does not raise the defect.

### Risk

It is expensive and inefficient to use the C-string output of a std::string function when you can use an std::string object instead. An std::string object contains the length of the string. When you use a C-string instead of an std::string object, the constructor determines the length of the C-string by a linear search, resulting in inefficient code. Using the C-string is also often unnecessary. Consider this code:

```
void set_prop1(const char* str);
void set_prop2(const std::string& str);
void foo( std::string& str){
    //...
    set_prop1(str.c_str()); // Necessary
    //...
    set_prop2(str.c_str()); // Inefficient
}
```

The function foo calls two different functions. Because the function set_prop1 requires a C-string as the input, using the str.c_str function is necessary to form the input to set_prop1. The function set_prop2 takes an std::string as an input. Instead of directly using str as an input to set_prop2, str.c_str is used, perhaps as a copy-paste mistake. The compiler implicitly constructs a new std::string object, which is identical to str, by using the output of str.c_str. Constructing a new std::string object in this case is unnecessary and inefficient. Because this code compiles and functions correctly, this inefficient code might not be noticed.

### Fix

To fix this defect, eliminate calls to std::string functions that produce a C-sting. Use std::string instead. Choose appropriate function overloads when you use a string object instead of a C-string. Consider this code:

```
void set_prop1(const char* str);
void set_prop2(const std::string& str);
void foo( std::string& input){
    //...
    set_prop1(str.c_str()); // Necessary
    //...
    set_prop2(str); // Efficient
}
```

Using str instead of str.c_str as input to set_prop2 makes the code more efficient and fixes the defect.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Implicit Construction of std::string Objects by Using C-String

```
#include <string>
#include <utility>

class A
{
public:
    A( char const* );
    char const* c_str() const;
};
void CppLibFuncA(const std::string&);
void CppLibFuncB(std::string &&);
void bar( A const& );
std::string make_string();
bool contains( std::string const& str1, std::string const& str2 )
{
    return str1.find( str2.data() ) == std::string::npos;
}

void foo(const std::string& s, std::string&& rs, A& other){
    CppLibFuncA( s.data() );
    CppLibFuncB( std::move( rs ).c_str() );
    CppLibFuncA( make_string().data() );
    bar( other.c_str() );
    if(contains(s,make_string())){
        //...
    }

}
```

In this example, Polyspace flags the implicit construction of a `string` object by using a C-string that is obtained from a `string` function.

- The function `CppLibFuncA` takes a `const std::string&` as input. When the function `CppLibFunc` is called by using `s.data()`, the compiler cannot pass a reference to the object `s` to the function. Instead, the compiler implicitly constructs a new `std::string` object from the C-string pointer and passes the new object to the function,which is inefficient. Polyspace flags the call to `std::string::data`.
- Because calling `CppLibFuncB` by using the output of `std::string::c_str` also implicitly constructs a new `str::string` object, Polyspace flags the call to `std::string::c_str`.
- A call to the function `bar` is not flagged because a `const char*` is not implicitly converted to a new `std::string` object.
- In the function `contain`, Polyspace flags the call to `std::string::find()`, where the output of `std::string::data` is used instead of an `std::string` object.

#### Correction

To fix this issue, avoid implicit construction of new `std::string` objects from the outputs of the `std::string::c_str` or `std::string::data` functions. Use the existing `std::string` objects instead.

```
#include <string>
#include <utility>
void CppLibFuncA(std::string const &);
void CppLibFuncB(std::string &&);
std::string make_string();
bool contains( std::string const& str1, std::string const& str2 )
{
    return str1.find( str2 ) == std::string::npos;
}
void foo(std::string const & s, std::string&& rs){
    CppLibFuncA( s );
    CppLibFuncB( std::move( rs ) );
    CppLibFuncA( make_string());
    if(contains(s,make_string())){
        //...
    }
}
```

Fix calls to the functions `CppLibFunc`, `CppLibFuncB`, and `CppLibFuncC` by using the existing `std::string` objects as input.

### Explicit Construction of `std::string` Objects by Using C-String

```
#include <string>
#include <utility>
std::string make_string(void);
void bar(const std::string& s){
    std::string s1 = s.c_str(); // Inefficient
    std::string s2 = make_string();
    s2.append(s1.data());
}
```

In this example, Polyspace flags the explicit use of a C-string when:

- The std::string object s is copied to s1 by calling s.c_str().
- The std::string object s1 is appended calling s1.data().

.

#### Correction

To fix this issue, avoid using a C-string when you can use an std::string object.

```
#include <string>
#include <string>
#include <utility>
std::string make_string(void);
void bar(const std::string& s){
    std::string s1 = s; // Efficient
    std::string s2 = make_string();
    s2.append(s1);
}
```

### Invoking std::string Member Functions by Using C-String

```
#include <string>
#include <utility>
std::string make_string(void);
void bar2( std::string& s1){
    std::string s2 = make_string();
    s1.replace(1, 1, &s2[0]);
    s1.replace(s1.begin(), s1.end(), &s2.at(0));
    s1.append(s2.c_str());
    s1.assign(s2.data());
    s1.compare(s2.data());
    const char* p = s2.c_str();
    s1.find(p);
}
```

In this example, Polyspace flags the explicit use of a C-string to invoke member functions of the std::string class.

#### Correction

To fix this issue, avoid using C-strings when you can use an std::string object instead.

```
#include <string>
#include <string>
#include <utility>
std::string make_string(void);
void bar( std::string& s1){
    std::string s2 = make_string();
    s1.replace(1, 1, s2);
    s1.replace(s1.begin(), s1.end(), s2);
    s1.append(s2);
    s1.assign(s2);
    s1.compare(s2);
    s1.find(s2);
}
```

#### Invoking User-Defined Functions by Using C-String

```
#include <string>
#include <utility>
void userDefined(const char* p){
    //...
}
void userDefined(const std::string& s){
    //...
    userDefined(s.c_str());//Compliant
    //userDefined(s);//Infinite recursion
}
void bar( const std::string& s){
    const char* p = s.data();
    userDefined(p);
    userDefined(s.c_str());
}
```

In this example, the user-defined function `userDefined` is overloaded to accept either a `const char*` or a `const std::string` parameter. Polyspace flags the use of a C-string instead of an `std::string` object to call the function. Polyspace does not flag calls to `std::string::c_str` or `std::string::data` when they are used for calling the `const char*` overload of `userDefined` from the body of the `std::string` overload of `userDefined`. In this case, using `std::string` instead of a C-string results in an unintended infinite recursion.

##### Correction

To fix this issue, avoid using C-strings when you can use an `std::string` object instead.

```
#include <string>
#include <string>
#include <utility>
extern void userDefined(const char *);
extern void userDefined(const std::string &);
void bar( const std::string& s){
    userDefined(s);
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_C_STR_STD_STRING_OPERATION`
**Impact:** Medium

# Version History

**Introduced in R2020b**

## See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"

"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Use of new or make_unique instead of more efficient make_shared

Using new or make_unique to initialize or reset shared_ptr results in additional memory allocation

## Description

This defect occurs when you use:

- new or make_unique to initialize a shared_ptr instance. For example:

  ```
  std::shared_ptr<T> p1(new T());
  std::shared_ptr<T> p2(make_unique<T>());
  ```
- new to reset a shared_ptr instance. For example:

  ```
  std::shared_ptr<T> p1;
  //...
  p1.reset(new T);
  ```

You use shared_ptr instances when you want multiple smart pointers to own and manage the same object. The instances also share a control block that holds a count of the number of instances that own the managed object.

Polyspace does not flag the use of new to initialize a shared_ptr instance in private or protected constructors. For example, no defect is raised on the use of new in this code snippet:

```
class PrivateCTor
{
public:
    static std::shared_ptr<PrivateCTor> makeOne()
    {
        return std::shared_ptr<PrivateCTor>(new PrivateCTor);
    }
private:
    PrivateCTor();
};
```

### Risk

When you use new or make_unique to initialize a shared_ptr instance, an additional allocation operation creates storage for the control block. The address of the additional allocation might be in a different memory page or outside the data-cache compared to the address of the managed object.

### Fix

Use std::make_shared to initialize a shared_ptr instance. The function performs a single allocation operation with enough memory to store the managed object and the control block.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Use of `new` to Initialize a `shared_ptr`

```
#include <memory>
#include <string>

struct Profile {
    virtual ~Profile() = default;
};

struct Player : public Profile {
    std::string name;
    std::int8_t rank;

    Player();
    Player(const std::string& name_, const std::int8_t& rank_) :
        name{ name_ }, rank{ rank_ } {}
};

void func()
{

    std::shared_ptr<Player> player1(new Player("Serena Williams", 1));
    std::shared_ptr<Player> top_rank(player1);

}
```

In this example, Polyspace flags the use of `new` to initialize `player1`. The program performs an additional allocation operation for the control block that holds the counter of all instances of `shared_ptr` that own the managed `Player` object.

#### Correction — Use `std::make_shared` to Initialize `shared_ptr`

One possible correction is to use `std::make_shared` to initialize `player1`.

```
#include <memory>
#include <string>

struct Profile {
    virtual ~Profile() = default;
};

struct Player : public Profile {
    std::string name;
    std::int8_t rank;

    Player();
    Player(const std::string& name_, const std::int8_t& rank_) :
        name{ name_ }, rank{ rank_ } {}
};

void func()
{

    auto player1 = std::make_shared<Player>("Serena Williams", 1);
    std::shared_ptr<Player> top_rank(player1);

}
```

## Result Information

**Group:** Performance

**Language:** C++
**Default:** Off
**Command-Line Syntax:** `MISSING_MAKE_SHARED`
**Impact:** Low

## Version History

**Introduced in R2021a**

### See Also

`Find defects (-checkers)` | `AUTOSAR C++14 Rule A20-8-6` | `AUTOSAR C++14 Rule A18-5-2`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive use of container's size method

A container's `size()` method is used for checking emptiness instead of its `empty()` method, which is more efficient

## Description

This defect occurs when you call a container's `size()` method to check if the container is empty. For instance:

```
std::list l;
//...
if(l.size()==0)// Inefficient
//..
```

when you use the method to check whether the container is empty, Polyspace flags `l.size()` .

### Risk

A container's `size()` method might be expensive if you use C++ version predating C++11. Using such a method to check for emptiness of a container is unnecessarily expensive and might result in inefficient code.

### Fix

To check a container's emptiness, use its `empty()` method instead. For instance:

```
std::list l;
//...
if(l.empty())// Efficient
//..
```

In C++ version predating C++11, the `empty()` methods of STL containers are `O(1)` but the `size()` method is `O(N)`. When checking for emptiness, the `empty()` method is the more efficient solution.

If you are using C++11 or later, the `size()` method of an STL container is as efficient as the `empty()` method. Because the `empty()` method clearly communicates the developers intent, it is a good practice to use it when checking if a container is empty.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Do Not Use Container's `size()` for Emptiness Check

```
#include <list>
#include <algorithm>
#include <string>

// is_container_empty
bool is_container_empty(const std::list<std::string> &mylist)
```

```
{
   return mylist.size() != 0;
}
```

In this example, the function `is_container_empty()` checks whether the list `mylist` is empty by comparing the output of `mylist.size()` to zero. This method of checking empliness of a container is inefficient. Polyspace flags the call to `mylist.size()`.

#### Correction — Use the Containers `empty()` Method

To resolve this defect, use the `empty()` method of the container when checking for emptiness.

```
#include <list>
#include <algorithm>
#include <string>

// is_container_empty
bool is_container_empty(const std::list<std::string> &mylist)
{
   return mylist.empty();
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_CONTAINER_EMPTINESS_CHECK`
**Impact:** Low

# Version History

**Introduced in R2022b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive use of map's bracket operator to insert or assign a value

The bracket operator of std::map or std::unordered_map is used for inserting or assigning a value in the container instead of the insert_or_assign() method, which is more efficient

## Description

This defect occurs when you use the [] operators to insert or assign a value to a key in a std::map or std::unordered_map instead of using the insert_or_assign() method. For instance:

```
std::map<int, std::string> Data;
Data[55] = "Fifty five"; //defect
```

Polyspace flags the use of the [] operator to insert a key value pair in the map data. This checker applies to code that uses C++17 or later.

### Risk

When you insert or assign a value in a std::map or std::unordered_map by using the [] operator, the compiler calls the default constructor of pair to constructs a key-value pair. The method insert_or_assign() inserts or assigns a value without calling the constructor, which makes it more efficient. Using the less efficient [] operator makes your code inefficient.

### Fix

When inserting or assigning a value in maps, use their insert_or_assign() method. For instance:

```
std::map<int, std::string> Data;
Data.insert_or_assign(55,"Fifty five");
```

The insert_or_assign() method inserts the key-value pair without having to construct a pair first, making it more efficient.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Avoid Using [] When Inserting a Value in a Map

```
#include <map>
#include <string>

void foo(std::map<std::string, std::string> &m, std::string name, std::string address)
{
    m[name] = address;
}
```

In this example, the value address is inserted against the key name in the map m. During this insertion a pair (name,address) is formed by calling the default constructor. This method of insertion into a map is inefficient and Polyspace flags it.

#### Correction — Use the Method `insert_or_assign()` Instead of `[]`

To fix this defect, use `insert_or_assign()` instead of `[]`.

```
#include <map>
#include <string>

void foo(std::map<std::string, std::string> &m, std::string name, std::string address)
{
    m.insert_or_assign(name,address);
}
```

### Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_MAP_INSERT_OR_ASSIGN`
**Impact:** Low

## Version History

**Introduced in R2022b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Expensive return caused by unnecessary std::move

An unnecessary call to `std::move` in the return statement hinders return value optimization, resulting in inefficient code

## Description

This defect occurs when you use `std::move` to force the compiler to return a local variable from a function by using move semantics. Polyspace reports a defect if the returned local variable satisfies all of these requirements:

- It is not a `volatile` object.
- It is not an input parameter of the function or a `catch` block.
- It has the same type as the return type of the function.
- It is returned by using `std::move`.

In this code, a local object is returned by using `std::move`. Polyspace flags the return statement.

```
class A{};
A func(A a){
A local_obj;
return std::move(local_obj);
}
```

The C++ standards define the necessary conditions when compilers are permitted to perform return value optimization. For more information, see copy elision.

### Risk

In C++, compilers can optimize return values by omitting the call to a copy or move constructor when returning objects. Using `std::move` to return certain local objects might stop the compiler from attempting this optimization, which results in inefficient code. This inefficiency might be difficult to diagnose because the code compiles and behaves correctly.

### Fix

Remove the call to `std::move` in return statements.

Performance improvements can vary based on your compiler, library implementation, and environment.

## Examples

### Avoid Calling `std::move` in Return Statements

```
#include<vector>
template<typename T>
std::vector<T> vectorFactory(size_t size)
```

```
{
    std::vector<T> bigdata;
    //...
    return std::move(bigdata);
}

class data{
    char* name;
    char* address;
    public:
    data():name(nullptr), address(nullptr){}
    //...
};
void foo(){
    std::vector<data> database = vectorFactory<data>(5);
}
```

In this example, the function template `vectorFactory` returns a `vector` of objects. Calling `std::move` in the return statement might hinder return value optimization, which results in inefficient code. Polyspace reports a defect.

#### Correction — Allow Return Value Optimization

To facilitate return value optimization by the compiler, avoid using `std:move` in the return statement.

```
#include<vector>
template<typename T>
std::vector<T> vectorFactory(size_t size)
{
    std::vector<T> bigdata;
    //...
    return bigdata;
}

class data{
    char* name;
    char* address;
    public:
    data():name(nullptr), address(nullptr){}
    //...
};
void foo(){
    std::vector<data> database = vectorFactory<data>(5);
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `EXPENSIVE_RETURN_STD_MOVE`
**Impact:** Low

# Version History

**Introduced in R2022b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing call to container's reserve method

A fixed number of items are added to a container without calling the `reserve()` method of the container beforehand, resulting in inefficient code

## Description

This defect occurs when you add a fixed number of items to a container without preallocating the necessary memory by calling the `reserve()` method of the container.

This code inserts three items into the vector `v` but does not preallocate the required memory by calling `v.reserve()`.

```
//...
std::vector< int > v;
    v.push_back( 0 );
    v.push_back( 1 );
    v.push_back( 2 );
```

Polyspace reports a defect.

### Risk

If you do not preallocate sufficient memory by calling the `reserve()` method of a container, insertion operations might allocate progressively larger chunks of memory, resulting in inefficient code. Allocating sufficient memory before the insertion operations allows for efficient use of memory and might allow for faster execution.

### Fix

Call the `reserve()` method of the container to preallocate memory. If you are able to use C++11, use `std::initializer_lists` to insert a known number of elements.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Insert Elements Without Preallocating Memory

```
#include <vector>

void fill_3() {
    std::vector< int > v;
    v.push_back( 0 ); /* Defect */
    v.push_back( 1 );
    v.push_back( 2 );
}
```

In this example, the function `fill_3` pushes three elements into a vector. Because the number of elements to be inserted is known, reserving memory for them beforehand might make the code more efficient. Polyspace reports a defect on the inefficient code.

#### Correction — Preallocate Necessary Memory

Call the `reserve()` method of the container to preallocate the necessary memory before the insertion operations.

```
#include <vector>

void fill_3() {
    std::vector< int > v;
    v.reserve( 3 );
    v.push_back( 0 ); /* Defect */
    v.push_back( 1 );
    v.push_back( 2 );
}
```

#### Correction — Use `std::Initializer_list`

If the number of elements to be inserted and their values are known, use an `initializer_list` to initiate the container. This solution applies to C++11 and later.

```
#include <vector>

void fill_3() {
    std::vector< int > v{0,1,2};
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `MISSING_CONTAINER_RESERVE`
**Impact:** Medium

# Version History

**Introduced in R2022b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unnecessary construction before reassignment

Instead of directly constructing objects with value, you construct objects and then immediately assign values to objects

## Description

Polyspace reports this defect if you construct a local object and then you immediately assign a value to the object in the same scope. For example:

```
class obj{/*...*/};
void foo(obj a){
    obj b;
    b = a; //Defect
}
```

This code is inefficient because you construct the object b and reassign a to b without using b. This is inefficient and Polyspace reports a defect. Polyspace flags the unnecessary construction of both trivial and nontrivial objects.

### Risk

Construction and immediate reassignment of an object requires two calls:

1. Call to the constructor of the object.
2. Call to the assignment operator.

When the assignment operator reconstructs the object, the code immediately discards the previously constructed value. The immediate reassignment makes the constructor call unnecessary. Because the code compiles and behaves correctly, you might not detect the unnecessary construction, making the code inefficient.

### Fix

To resolve this defect, directly construct the object with the value by using the copy or move constructor. For instance:

```
class obj{/*...*/};
void foo(obj a){
    obj b{a};

}
```

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Inefficient Construction of Expensive Object

In this example, the functions `func1()` and `func2` construct an object and then immediately reassign the value. Using two statements for construction and reassignment is inefficient. Polyspace reports defects on the reassignment statements.

```
#include <string>
#include <unordered_set>
#include <vector>

typedef std::unordered_set<std::string> Type1;
typedef std::vector<std::string> Type2;

template< typename T >
T factory();

void func1(){
    Type1 obj;
    obj = factory< Type1 >(); // Defect
}

void func2(){
    Type2 obj;
    obj = factory< Type2 >(); // Defect
}
```

#### Correction — Construct Objects by Using Value

To fix this defect, combine the constructor and assignment statements into one statement:

```
#include <string>
#include <unordered_set>
#include <vector>

typedef std::unordered_set<std::string> Type1;
typedef std::vector<std::string> Type2;

template< typename T >
T factory();

void func1(){
    Type1 obj = factory< Type1 >(); //No defect
}

void func2(){
    Type2 obj = factory< Type2 >(); //No defect
}
```

## Result Information

**Group:** Performance
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `UNNECESSARY_CONSTRUCTION_BEFORE_ASSIGNMENT`
**Impact:** Low

## Version History

**Introduced in R2023a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unnecessary implementation of a special member function

Implementing special member functions hinders code optimization when implicit versions are equivalent

## Description

Polyspace reports this defect if you implement a special member function in your class with the same functionalities as the implicit version of the function. The implicit special member functions typically have these properties:

- The implicit default constructor has an empty body, an empty parameter list, and an empty initializer list.
- The implicit destructor has an empty body.
- The implicit copy constructor copies the base classes and nonstatic data members by using an initializer list. The implicit move constructor moves the base classes and nonstatic data members using an initializer list. The implicit copy and move constructors do not create deep copies and do not move the data associated with a raw pointer.
- The implicit copy and move assignment operators use base and member-wise assignments.

For details about how implicitly defined special member functions behave, see:

- Default constructor
- Destructor
- Copy constructor
- Copy assignment operator
- Move constructor
- Move assignment operator

### Risk

Compilers optimize your code by turning implicit special member functions into trivial functions, which are faster to execute. This optimization does not take place if you implement a special member function in your code. Implementing a special member function that you can omit or declare as `=default` results in slower code.

Unnecessary implementation of a special member function makes your code more complex, hinders optimization, and makes the code harder to read and maintain.

### Fix

To fix this defect, take one of these actions:

- Omit the implementation of the flagged special member function.
- Declare the special member function as `=default`.

Performance improvements might vary based on the compiler, library implementation, and environment that you are using.

## Examples

### Unnecessary Copy and Move Constructors

In this example, you define the class A. In this class:

- Because the user-defined copy and move constructors of A do not manage any resources and they can be defined as =default or left undefined, Polyspace reports defects for these functions. Leaving these functions undeclared or declaring them as =default allows the compiler to turn these operations into trivial operations, resulting in more efficient code.
- The user-defined default constructor has a nonempty initializer list. Because this default constructor initializes two data members, an implicit default constructor cannot replicate its functionalities. Similarly, an implicit constructor cannot replace the nondefault constructor. Polyspace does not report defects for these constructors.

```
#include <cstdint>
#include <utility> //for std::move
class A
{
public:
    A() : x(0), y(0) {}                                   // Not a defect
    A(std::int32_t first, std::int32_t second)            // Not a defect
    : x(first), y(second) {}
    A(const A& oth) : x(oth.x),y(oth.y){}                 // Defect
    A(A&& oth): x(std::move(oth.x)),y(std::move(oth.y)){} // Defect
    ~A(){}                                                // Defect


private:
    std::int32_t x;
    std::int32_t y;
};
```

#### Correction — Avoid Implementing Unnecessary Special Member Functions

To resolve the defects, declare the flagged special member functions a =default.

```
#include <cstdint>
#include <utility>
class A
{
public:
    A() : x(0), y(0) {}                                   // Not a defect
    A(std::int32_t first, std::int32_t second)            // Not a defect
    : x(first), y(second) {}
    A(const A& oth)   =default;                           // Resolved
    A(A&& oth)=default;                                   // Resolved
    ~A()=default;                                         // Resolved


private:
    std::int32_t x;
    std::int32_t y;
};
```

### Necessary Implementation of Special Member Function

When your class manages resources such as a raw pointer or a POSIX file descriptor, the implicit special member functions are typically insufficient. In such cases, to avoid unexpected behavior, the best practice is to implement all the special member functions. In this example, the class A manages a raw pointer *p. To avoid unexpected behavior, implement special member functions that perform memory operations. For instance, the copy constructor in this code performs a deep copy and the move constructor invokes std::move(). Because these member functions perform actions that the implicit member functions do not perform, Polyspace does not report a defect.

```
#include <iostream>
#include<utility>
class A{
    private:
    int* p;
    public:
    A(){
        p = new int;
    }
    A(int in){
        p = new int;
        *p = in;
    }
    ~A(){
        delete p;
    }
    A(const A& oth){
        p = new int;
        *p = *(oth.p);
    }
    A(A&& oth){
        p = new int;
        *p = std::move(*oth.p);
    }
};
```

### Default Construction of const Instances

In this example, the function foo() constructs a const instance of myclass. Because the code does not initialize myclass::x in the class, the implicit default constructor cannot correctly construct a const instance myclass. When you use compilers such as gcc or clang, declaring myclass()=default; results in a compile fail.

Providing a user-defined constructor that is identical to the implicit constructor to resolve this issue is inefficient. In this code, the constructor myclass::myclass() is identical to an implicit constructor. This constructor resolves the compilation issue, but this solution is inefficient and Polyspace reports a defect.

```
class myclass{
    public:
    //myclass()=default;// Compile fail
    myclass(){}         //Defect
    int x;
};

void foo(){
```

```
const myclass obj;
}
```

#### Correction — Refactor myclass to Use Implicit Constructor

To allow implicit default construction of `const` instances of `myclass`, initialize `myclass::x` in the class definitions.

```
class myclass{
    public:
    myclass()=default;// Resolved
    int x=0;
};

void foo(){
const myclass obj;
}
```

### Result Information

**Group:** Performance
**Language:** C++ (C++11 or later)
**Default:** Off
**Command-Line Syntax:** `UNNECESSARY_IMPL_OF_SPECIAL_MEMBER_FUNCTION`
**Impact:** Low

## Version History

**Introduced in R2023a**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# 19

# Resource Management Defects

# Closing previously closed resource

Function closes a previously closed stream

## Description

This defect occurs when a function attempts to close a stream that was closed earlier in your code and not reopened later.

### Risk

The standard states that the value of a `FILE*` pointer is indeterminate after you close the stream associated with it. Performing the close operation on the `FILE*` pointer again can cause unwanted behavior.

### Fix

Remove the redundant close operation.

## Examples

### Closing Previously Closed Resource

```
#include <stdio.h>

void func(char* data) {
    FILE* fp = fopen("file.txt", "w");
    if(fp!=NULL) {
        if(data)
            fputc(*data,fp);
        else
            fclose(fp);
    }
    fclose(fp);
}
```

In this example, if `fp` is not `NULL` and `data` is `NULL`, the `fclose` operation occurs on `fp` twice in succession.

#### Correction — Remove Close Operation

One possible correction is to remove the last `fclose` operation. To avoid a resource leak, you must also place an `fclose` operation in the `if(data)` block.

```
#include <stdio.h>

void func(char* data) {
    FILE* fp = fopen("file.txt", "w");
    if(fp!=NULL) {
        if(data) {
            fputc(*data,fp);
            fclose(fp);
        }
        else
            fclose(fp);
    }
}
```

### Result Information

**Group:** Resource management
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `DOUBLE_RESOURCE_CLOSE`
**Impact:** High

## Version History

**Introduced in R2015b**

### See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Opening previously opened resource

Opening an already opened file

## Description

This defect occurs when a file handling function such as `fopen` opens a file that was previously opened and not closed subsequently.

### Risk

If you open a resource multiple times, you can encounter:

- A race condition when accessing the file.
- Undefined or unexpected behavior for that file.
- Portability issues when you run your program on different targets.

### Fix

Once a resource is open, close the resource before reopening.

## Examples

### File Reopened With New Permissions

```
#include <stdio.h>
const char* logfile = "my_file.log";

void doubleresourceopen()
{
    FILE* fpa = fopen(logfile, "w");
    if (fpa == NULL) {
        return;
    }
    (void)fprintf(fpa, "Writing");
    FILE* fpb = fopen(logfile, "r");
    (void)fclose(fpa);
    (void)fclose(fpb);
}
```

In this example, a `logfile` is opened in the first line of this function with write privileges. Halfway through the function, the `logfile` is opened again with read privileges.

#### Correction — Close Before Reopening

One possible correction is to close the file before reopening the file with different privileges.

```
#include <stdio.h>
const char* logfile = "my_file.log";

void doubleresourceopen()
```

```
{
    FILE* fpa = fopen(logfile, "w");
    if (fpa == NULL) {
        return;
    }
    (void)fprintf(fpa, "Writing");
    (void)fclose(fpa);
    FILE* fpb = fopen(logfile, "r");
    (void)fclose(fpb);
}
```

## Result Information

**Group:** Resource management
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `DOUBLE_RESOURCE_OPEN`
**Impact:** Medium

# Version History

**Introduced in R2016b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Resource leak

File stream not closed before FILE pointer scope ends or pointer is reassigned

## Description

This defect occurs when you open a file stream by using a FILE pointer but do not close it before:

- The end of the pointer’s scope.
- Assigning the pointer to another stream.

### Risk

If you do not release file handles explicitly as soon as possible, a failure can occur due to exhaustion of resources.

### Fix

Close a FILE pointer before the end of its scope, or before you assign the pointer to another stream.

## Examples

### FILE Pointer Not Released Before End of Scope

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

In this example, the file pointer fp1 is pointing to a file data1.txt. Before fp1 is explicitly dissociated from the file stream of data1.txt, it is used to access another file data2.txt.

#### Correction — Release FILE Pointer

One possible correction is to explicitly dissociate fp1 from the file stream of data1.txt.

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );
    fclose(fp1);

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

## Result Information

**Group:** Resource management
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** RESOURCE_LEAK
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Use of previously closed resource

Function operates on a previously closed stream

## Description

This defect occurs when a function operates on a stream that you closed earlier in your code.

### Risk

The standard states that the value of a `FILE*` pointer is indeterminate after you close the stream associated with it. Operations using the `FILE*` pointer can produce unintended results.

### Fix

One possible fix is to close the stream only at the end of operations. Another fix is to reopen the stream before using it again.

## Examples

### Use of FILE* Pointer After Closing Stream

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fclose(fp);
        fprintf(fp,"text");
    }
}
```

In this example, `fclose` closes the stream associated with `fp`. When you use `fprintf` on `fp` after `fclose`, the **Use of previously closed resource** defect appears.

#### Correction — Close Stream After All Operations

One possible correction is to reverse the order of the `fprintf` and `fclose` operations.

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fprintf(fp,"text");
        fclose(fp);
    }
}
```

## Result Information

**Group:** Resource management
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `CLOSED_RESOURCE_USE`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)` | `MISRA C:2012 Rule 22.6`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Writing to read-only resource

File initially opened as read only is modified

## Description

This defect occurs when you attempt to write to a file that you have opened earlier in read-only mode.

For instance, you open a file using `fopen` with the access mode argument `r`. You write to that file with a function in the `fprintf` family.

### Risk

Writing to a read-only file causes undefined behavior.

### Fix

If you want to write to the file, open the file in a mode that is suitable for writing.

## Examples

### Writing to Read-Only File

```
#include <stdio.h>

void func(void) {
    FILE* fp ;

    fp = fopen("file.txt", "r");
    fprintf(fp, "Some data");
    fclose(fp);
}
```

In this example, the file `file.txt` is opened in read-only mode. When the `FILE` pointer associated with `file.txt` is used as an argument of `fprintf`, a **Writing to read-only resource** defect occurs.

#### Correction — Open File as Writable

One possible correction is to use the access specifier `"a"` instead of `"r"`. `file.txt` is now open for output at the end of the file.

```
#include <stdio.h>

void func(void) {
    FILE* fp ;

    fp = fopen("file.txt", "a");
    fprintf(fp, "Some data");
    fclose(fp);
}
```

## Result Information

**Group:** Resource management
**Language:** C | C++
**Default:** On for handwritten code, off for generated code
**Command-Line Syntax:** `READ_ONLY_RESOURCE_WRITE`
**Impact:** High

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

20

# Good Practice Defects

# Ambiguous declaration syntax

Declaration syntax can be interpreted as object declaration or part of function declaration

## Description

This defect occurs when it is not clear from a declaration whether an object declaration or function/ parameter declaration is intended. The ambiguity is often referred to as most vexing parse.

For instance, these declarations are ambiguous:

- `ResourceType aResource();`

  It is not immediately clear if `aResource` is a function returning a variable of type `ResourceType` or an object of type `ResourceType`.
- `TimeKeeper aTimeKeeper(Timer());`

  It is not immediately clear if `aTimeKeeper` is an object constructed with an unnamed object of type `Timer` or a function with an unnamed function pointer type as parameter. The function pointer refers to a function with no argument and return type `Timer`.

The checker does not flag ambiguous declarations with global scope. For instance, the analysis does not flag declarations with global scope using the format *`Type`* `a()` where *`Type`* is a class type with a default constructor. The analysis interprets `a` as a function returning the type *`Type`*.

### Risk

In case of an ambiguous declaration, the C++ Standard chooses a specific interpretation of the syntax. For instance:

- `ResourceType aResource();`

  is interpreted as a declaration of a function `aResource`.
- `TimeKeeper aTimeKeeper(Timer());`

  is interpreted as a declaration of a function `aTimeKeeper` with an unnamed parameter of function pointer type.

If you or another developer or code reviewer expects a different interpretation, the results can be unexpected.

For instance, later you might face a compilation error that is difficult to understand. Since the default interpretation indicates a function declaration, if you use the function as an object, compilers might report a compilation error. The compilation error indicates that a conversion from a function to an object is being attempted without a suitable constructor.

### Fix

Make the declaration unambiguous. For instance, fix these ambiguous declarations as follows:

- `ResourceType aResource();`

*Object declaration*:

If the declaration refers to an object initialized with the default constructor, rewrite it as:

```
ResourceType aResource;
```

prior to C++11, or as:

```
ResourceType aResource{};
```

after C++11.

*Function declaration*:

If the declaration refers to a function, use a typedef for the function.

```
typedef ResourceType(*resourceFunctionType)();
resourceFunctionType aResource;
```

- `TimeKeeper aTimeKeeper(Timer());`

  *Object declaration*:

  If the declaration refers to an object `aTimeKeeper` initialized with an unnamed object of class `Timer`, add an extra pair of parenthesis:

  ```
  TimeKeeper aTimeKeeper( (Timer()) );
  ```

  prior to C++11, or use braces:

  ```
  TimeKeeper aTimeKeeper{Timer{}};
  ```

  after C++11.

  *Function declaration*:

  If the declaration refers to a function `aTimeKeeper` with a unnamed parameter of function pointer type, use a named parameter instead.

  ```
  typedef Timer(*timerType)();
  TimeKeeper aTimeKeeper(timerType aTimer);
  ```

## Examples

### Function or Object Declaration

```
class ResourceType {
      int aMember;
    public:
      int getMember();
};

void getResource() {
    ResourceType aResource();
}
```

In this example, `aResource` might be used as an object but the declaration syntax indicates a function declaration.

#### Correction — Use {} for Object Declaration

One possible correction (after C++11) is to use braces for object declaration.

```
class ResourceType {
      int aMember;
    public:
      int getMember();
};

void getResource() {
    ResourceType aResource{};
}
```

### Unnamed Object or Unnamed Function Parameter Declaration

```
class MemberType {};

class ResourceType {
      MemberType aMember;
    public:
      ResourceType(MemberType m) {aMember = m;}
      int getMember();
};

void getResource() {
    ResourceType aResource(MemberType());
}
```

In this example, `aResource` might be used as an object initialized with an unnamed object of type `MemberType` but the declaration syntax indicates a function with an unnamed parameter of function pointer type. The function pointer points to a function with no arguments and type `MemberType`.

#### Correction — Use {} for Object Declaration

One possible correction (after C++11) is to use braces for object declaration.

```
class MemberType {};

class ResourceType {
      MemberType aMember;
    public:
      ResourceType(MemberType m) {aMember = m;}
      int getMember();
};

void getResource() {
    ResourceType aResource{MemberType()};
}
```

### Unnamed Object or Named Function Parameter Declaration

```
class Integer {
    int aMember;
public:
    Integer(int d) {aMember = d;}
    int getMember();
};

int aInt = 0;
void foo(){
    Integer aInteger(Integer(aInt));
}
```

In this example, `aInteger` might be an object constructed with an unnamed object `Integer(aInt)` (an object of class `Integer` which itself is constructed using the variable `aInt`). However, the declaration syntax indicates that `aInteger` is a function with a named parameter `aInt` of type `Integer` (the superfluous parenthesis is ignored).

**Correction — Use of {} for Object Declaration**

One possible correction (after C++11) is to use {} for object declaration.

```
class Integer {
    int aMember;
public:
    Integer(int d) {aMember = d;}
    int getMember();
};

int aInt = 0;
void foo(){
    Integer aInteger(Integer{aInt});
}
```

**Correction — Remove Superfluous Parenthesis for Named Parameter Declaration**

If `aInteger` is a function with a named parameter `aInt`, remove the superfluous () around `aInt`.

```
class Integer {
    int aMember;
public:
    Integer(int d) {aMember = d;}
    int getMember();
};

int aInt = 0;
void foo(){
    Integer aInteger(Integer aInt);
}
```

## Result Information

**Group:** Good practice
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `MOST_VEXING_PARSE`
**Impact:** Low

## Version History

**Introduced in R2019a**

## See Also

`Variable shadowing` | `Non-initialized variable` | `Write without a further read` | `Improper array initialization` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Bitwise and arithmetic operations on the same data

Statement with mixed bitwise and arithmetic operations

## Description

This defect occurs when bitwise and arithmetic operations are performed in the same expression.

### Risk

Mixed bitwise and arithmetic operations *do* compile. However, the size of integer types affects the result of these mixed operations. For instance, the arithmetic equivalent of a left shift (<<) by a certain number of bits depends on the number of bits in the variable being shifted and therefore on the internal representation of its data type. With a mix of bitwise and arithmetic operations, the same expression can produce different results on different targets.

Mixed operations also reduce readability and maintainability.

### Fix

Separate bitwise and arithmetic operations, or use only one type of operation per statement.

## Examples

### Shift and Addition

```
unsigned int bitwisearithmix()
{
    unsigned int var = 50;
    var += (var << 2) + 1;
    return var;
}
```

This example shows bitwise and arithmetic operations on the variable `var`. `var` is shifted by two (bitwise), then increased by 1 and added to itself (arithmetic).

#### Correction — Arithmetic Operations Only

You can reduce this expression to arithmetic-only operations: `var + (var << 2)` is equivalent to `var * 5`.

```
unsigned int bitwisearithmix()
{
    unsigned int var = 50;
    var = var * 5 +1;
    return var;
}
```

## Result Information

**Group:** Good Practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `BITWISE_ARITH_MIX`
**Impact:** Low

# Version History

**Introduced in R2016b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# C++ reference to const-qualified type with subsequent modification

Reference to `const`-qualified type is subsequently modified

## Description

This defect occurs when a variable that refers to a `const`-qualified type is modified after declaration.

For instance, in this example, `refVal` has a type `const int &`, but its value is modified in a subsequent statement.

```
using constIntRefType = const int &;
void func(constIntRefType refVal, int val){
   ...
   refVal = val; //refVal is modified
   ...
}
```

### Risk

The `const` qualifier on a reference type implies that a variable of the type is initialized at declaration and will not be subsequently modified.

Compilers can detect modification of references to `const`-qualified types as a compilation error. If the compiler does not detect the error, the behavior is undefined.

### Fix

Avoid modification of `const`-qualified reference types. If the modification is required, remove the `const` qualifier from the reference type declaration.

## Examples

### Modification of `const`-qualified Reference Types

```
typedef const int cint;
typedef cint& ref_to_cint;

void func(ref_to_cint refVal, int initVal){
   refVal = initVal;
}
```

In this example, `ref_to_cint` is a reference to a `const`-qualified type. The variable `refVal` of type `ref_to_cint` is supposed to be initialized when `func` is called and not modified subsequently. The modification violates the contract implied by the `const` qualifier.

#### Correction — Avoid Modification of `const`-qualified Reference Types

One possible correction is to avoid the `const` in the declaration of the reference type.

```
typedef int& ref_to_int;
```

```
void func(ref_to_int refVal, int initVal){
   refVal = initVal;
}
```

## Result Information

**Group:** Good practice
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `WRITE_REFERENCE_TO_CONST_TYPE`
**Impact:** Low

# Version History

**Introduced in R2019a**

## See Also

`C++ reference type qualified with const or volatile` | `Qualifier removed in conversion` | `Writing to const qualified object` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# C++ reference type qualified with const or volatile

Reference type declared with a redundant `const` or `volatile` qualifier

## Description

This defect occurs when a variable with reference type is declared with the `const` or `volatile` qualifier, for instance:

```
char &const c;
```

### Risk

The C++14 Standard states that `const` or `volatile` qualified references are ill formed (unless they are introduced through a `typedef`, in which case they are ignored). For instance, a reference to one variable cannot be made to refer to another variable. Therefore, using the `const` qualifier is not required for a variable with a reference type.

Often the use of these qualifiers indicate a coding error. For instance, you meant to declare a reference to a `const`-qualified type:

```
char const &c;
```

but instead declared a `const`-qualified reference:

```
char &const c;
```

If your compiler does not detect the error, you can see unexpected results. For instance, you might expect `c` to be immutable but see a different value of `c` compared to its value at declaration.

### Fix

See if the `const` or `volatile` qualifier is incorrectly placed. For instance, see if you wanted to refer to a `const`-qualified type and entered:

```
char &const c;
```

instead of:

```
char const &c;
```

If the qualifier is incorrectly placed, fix the error. Place the `const` or `volatile`qualifier before the & operator. Otherwise, remove the redundant qualifier.

## Examples

### `const`-Qualified Reference Type

```
int func (int &const iRef) {
    iRef++;
    return iRef%2;
}
```

In this example, `iRef` is a `const`-qualified reference type. Since `iRef` cannot refer to another variable, the `const` qualifier is redundant.

#### Correction — Remove `const` Qualifier

Remove the redundant `const` qualifier. Since `iRef` is modified in `func`, it is not meant to refer to a `const`-qualified variable. Moving the `const` qualifier before & will cause a compilation error.

```
int func (int &iRef) {
    iRef++;
    return iRef%2;
}
```

#### Correction — Fix Placement of `const` Qualifier

If you do not identify to modify `iRef` in `func`, declare `iRef` as a reference to a `const`-qualified variable. Place the `const` qualifier before the & operator. Make sure you do not modify `iRef` in `func`.

```
int func (int const &iRef) {
    return (iRef+1)%2;
}
```

### Result Information

**Group:** Good practice
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `CV_QUALIFIED_REFERENCE_TYPE`
**Impact:** Low

## Version History

**Introduced in R2019a**

### See Also

`Qualifier removed in conversion` | `Unreliable cast of pointer` | `Unreliable cast of function pointer` | `C++ reference to const-qualified type with subsequent modification` | `Writing to const qualified object` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Delete of void pointer

`delete` operates on a `void*` pointer pointing to an object

## Description

This defect occurs when the `delete` operator operates on a `void*` pointer.

### Risk

Deleting a `void*` pointer is undefined according to the C++ Standard.

If the object is of type `MyClass` and the `delete` operator operates on a `void*` pointer pointing to the object, the `MyClass` destructor is not called.

If the destructor contains cleanup operations such as release of resources or decreasing a counter value, the operations do not take place.

### Fix

Cast the `void*` pointer to the appropriate type. Perform the `delete` operation on the result of the cast.

For instance, if the `void*` pointer points to a `MyClass` object, cast the pointer to `MyClass*`.

## Examples

### Delete of void* Pointer

```
#include <iostream>

class MyClass {
public:
    explicit MyClass(int i):m_i(i) {}
    ~MyClass() {
        std::cout << "Delete MyClass(" << m_i << ")" << std::endl;
    }
private:
    int m_i;
};

void my_delete(void* ptr) {
    delete ptr;
}


int main() {
    MyClass* pt = new MyClass(0);
    my_delete(pt);
    return 0;
}
```

In this example, the function my_delete is designed to perform the delete operation on any type. However, in the function body, the delete operation acts on a void* pointer, ptr. Therefore, when you call my_delete with an argument of type MyClass, the MyClass destructor is not called.

#### Correction — Cast void* Pointer to MyClass*

One possible solution is to use a function template instead of a function for my_delete.

```
#include <iostream>

class MyClass {
public:
    explicit MyClass(int i):m_i(i) {}
    ~MyClass() {
        std::cout << "Delete MyClass(" << m_i << ")" << std::endl;
    }
private:
    int m_i;
};


template<typename T> void safe_delete(T*& ptr) {
    delete ptr;
    ptr = NULL;
}


int main() {
    MyClass* pt = new MyClass(0);
    safe_delete(pt);
    return 0;
}
```

## Result Information

**Group:** Good practice
**Language:** C++
**Default:** Off
**Command-Line Syntax:** DELETE_OF_VOID_PTR
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

Find defects (-checkers)

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Hard-coded buffer size

Size of memory buffer is a numerical value instead of symbolic constant

## Description

This defect occurs when you use a numerical value instead of a symbolic constant when declaring a memory buffer such as an array.

### Risk

Hard-coded buffer size causes the following issues:

- Hard-coded buffer size increases the likelihood of mistakes and therefore maintenance costs. If a policy change requires developers to change the buffer size, they must change every occurrence of the buffer size in the code.
- Hard-constant constants can be exposed to attack if the code is disclosed.

### Fix

Use a symbolic name instead of a hard-coded constant for buffer size. Symbolic names include `const`-qualified variables, `enum` constants, or macros.

`enum` constants are recommended.

- Macros are replaced by their constant values after preprocessing. Therefore, they can expose the loop boundary.
- `enum` constants are known at compilation time. Therefore, compilers can optimize the loops more efficiently.

  `const`-qualified variables are usually known at run time.

## Examples

### Hard-Coded Buffer Size

```
int table[100];

void read(int);

void func(void) {
    for (int i=0; i<100; i++)
        read(table[i]);
}
```

In this example, the size of the array `table` is hard-coded.

#### Correction — Use Symbolic Name

One possible correction is to replace the hard-coded size with a symbolic name.

```
const int MAX_1 = 100;
#define MAX_2 100
enum { MAX_3 = 100 };

int table_1[MAX_1];
int table_2[MAX_2];
int table_3[MAX_3];

void read(int);

void func(void) {
    for (int i=0; i < MAX_1; i++)
        read(table_1[i]);
    for (int i=0; i < MAX_2; i++)
        read(table_2[i]);
    for (int i=0; i < MAX_3; i++)
        read(table_3[i]);
}
```

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `HARD_CODED_BUFFER_SIZE`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Hard-coded loop boundary

Loop boundary is a numerical value instead of symbolic constant

## Description

This defect occurs when you use a numerical value instead of symbolic constant for the boundary of a `for`, `while` or `do-while` loop.

### Risk

Hard-coded loop boundary causes the following issues:

- Hard-coded loop boundary makes the code vulnerable to denial of service attacks when the loop involves time-consuming computation or resource allocation.
- Hard-coded loop boundary increases the likelihood of mistakes and maintenance costs. If a policy change requires developers to change the loop boundary, they must change every occurrence of the boundary in the code.

  For instance, the loop boundary is 10000 and represents the maximum number of client connections supported in a network server application. If the server supports more clients, you must change all instances of the loop boundary in your code. Even if the loop boundary occurs once, you have to search for a numerical value of `10000` in your code. The numerical value can occur in places other than the loop boundary. You must browse through those places before you find the loop boundary.

### Fix

Use a symbolic name instead of a hard-coded constant for loop boundary. Symbolic names include `const`-qualified variables, `enum` constants or macros.`enum` constants are recommended because:

- Macros are replaced by their constant values after preprocessing. Therefore, they can expose the buffer size.
- `enum` constants are known at compilation time. Therefore, compilers can allocate storage for them more efficiently.

  `const`-qualified variables are usually known at run time.

## Examples

### Hard-Coded Loop Boundary

```
void performOperation(int);

void func(void) {
    for (int i=0; i<100; i++)
        performOperation(i);
}
```

In this example, the boundary of the `for` loop is hard-coded.

#### Correction — Use Symbolic Name

One possible correction is to replace the hard-coded loop boundary with a symbolic name.

```
const int MAX_1 = 100;
#define MAX_2 100
enum { MAX_3 = 100 };

void performOperation_1(int);
void performOperation_2(int);
void performOperation_3(int);

void func(void) {
    for (int i=0; i<MAX_1; i++)
        performOperation_1(i);
    for (int i=0; i<MAX_2; i++)
        performOperation_2(i);
    for (int i=0; i<MAX_3; i++)
        performOperation_3(i);
}
```

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `HARD_CODED_LOOP_BOUNDARY`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Hard-coded object size used to manipulate memory

Memory manipulation with hard-coded size instead of `sizeof`

## Description

This defect occurs when you use hard-coded constants as memory size arguments for these memory functions:

- Dynamic memory allocation function such as `malloc` or `calloc`.
- Memory manipulation functions such as `memcpy`, `memmove`, `memcmp`, or `memset`.

When performing memory operations with a string literal, Polyspace does not report a defect if you hard code the memory size.

### Risk

If you hard code object size, your code is not portable to architectures with different type sizes. If the constant value is not the same as the object size, the buffer might or might not overflow.

### Fix

For the size argument of memory functions, use `sizeof(`*`object`*`)`.

## Examples

### Assume 4-Byte Integer Pointers

```
#include <stddef.h>
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};
extern void fill_ints(int **matrix, size_t nb, size_t s);

void bug_hardcodedmemsize()
{
    size_t i, s;

    s = 4;
    int **matrix = (int **)calloc(SIZE20, s);
    if (matrix == NULL) {
        return; /* Indicate calloc() failure */
    }
    fill_ints(matrix, SIZE20, s);
    free(matrix);
}
```

In this example, the memory allocation function `calloc` is called with a memory size of 4. The memory is allocated for an integer pointer, which can be a more or less than 4 bytes depending on your target. If the integer pointer is not 4 bytes, your program can fail.

#### Correction — Use `sizeof(int *)`

When calling `calloc`, replace the hard-coded size with a call to `sizeof`. This change makes your code more portable.

```
#include <stddef.h>
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};
extern void fill_ints(int **matrix, size_t nb, size_t s);

void corrected_hardcodedmemsize()
{
    size_t i, s;

    s = sizeof(int *);
    int **matrix = (int **)calloc(SIZE20, s);
    if (matrix == NULL) {
        return; /* Indicate calloc() failure */
    }
    fill_ints(matrix, SIZE20, s);
    free(matrix);
}
```

### Hard Coded Size in `memset`

In this example, the function `clean_sensitive_memory` clears sensitive information from the memory. Here, the memory size argument of `memset` is hardcoded to be 64 bytes. If `s->data` cannot accommodate `64` bytes, the program fails and the sensitive information might remain in memory. Polyspace reports defects on the memory operation.

```
#include<string.h>

struct sensitiveInfo
{
    unsigned char data[64];
    int length;
};

char key[64];

void clean_sensitive_memory (struct sensitiveInfo *s)
{
    memset (s->data, 0, 64);          //Defect
    memset ((void *) key, 0, 64);  //Defect
}
```

#### Correction — Use `sizeof()` for Size Argument

To fix this defect, replace the hardcoded memory sizes by calls to `sizeof()`.

```
#include<string.h>

struct sensitiveInfo
{
    unsigned char data[64];
    int length;
};

char key[64];

void clean_sensitive_memory (struct sensitiveInfo *s)
{
  memset (s->data, 0, sizeof (s->data));    //Fixed
  memset ((void *) key, 0, sizeof(key)); //Fixed
}
```

## Result Information

**Group:** Good Practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `HARD_CODED_MEM_SIZE`
**Impact:** Low

# Version History

**Introduced in R2016b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Incorrect syntax of flexible array member size

Flexible array member defined with size zero or one

## Description

This defect occurs when you do not use the standard C syntax to define a structure with a flexible array member.

Since C99, you can define a flexible array member with an unspecified size. For instance, `desc` is a flexible array member in this example:

```
struct record {
    size_t len;
    double desc[];
};
```

Prior to C99, you might have used compiler-specific methods to define flexible arrays. For instance, you used arrays of size one or zero:

```
struct record {
    size_t len;
    double desc[0];
};
```

This usage is not compliant with the C standards following C99.

### Risk

If you define flexible array members by using size zero or one, your implementation is compiler-dependent. For compilers that do not recognize the syntax, an `int` array of size one has buffer for one `int` variable. If you try to write beyond this buffer, you can run into issues stemming from array access out of bounds.

If you use the standard C syntax to define a flexible array member, your implementation is portable across all compilers conforming with the standard.

### Fix

To implement a flexible array member in a structure, define an array of unspecified size. The structure must have one member besides the array and the array must be the last member of the structure.

## Examples

### Flexible Array Member Defined with Size One

```
#include <stdlib.h>

struct flexArrayStruct {
  int num;
  int data[1];
};

unsigned int max_size = 100;

void func(unsigned int array_size) {
  if(array_size<= 0 || array_size > max_size)
      exit(1);
  /* Space is allocated for the struct */
  struct flexArrayStruct *structP
    = (struct flexArrayStruct *)
     malloc(sizeof(struct flexArrayStruct)
          + sizeof(int) * (array_size - 1));
  if (structP == NULL) {
    /* Handle malloc failure */
    exit(2);
  }

  structP->num = array_size;

  /*
   * Access data[] as if it had been allocated
   * as data[array_size].
   */
  for (unsigned int i = 0; i < array_size; ++i) {
    structP->data[i] = 1;
  }

  free(structP);
}
```

In this example, the flexible array member `data` is defined with a size value of one. Compilers that do not recognize this syntax treat `data` as a size-one array. The statement `structP->data[i] = 1;` can write to `data` beyond the first array member and cause out of bounds array issues.

#### Correction — Use Standard C Syntax to Define Flexible Array

Define flexible array members with unspecified size.

```
#include <stdlib.h>

struct flexArrayStruct{
  int num;
  int data[];
};

unsigned int max_size = 100;

void func(unsigned int array_size) {
  if(array_size<=0 || array_size > max_size)
      exit(1);

  /* Allocate space for structure */
  struct flexArrayStruct *structP
    = (struct flexArrayStruct *)
    malloc(sizeof(struct flexArrayStruct)
         + sizeof(int) * array_size);

  if (structP == NULL) {
    /* Handle malloc failure */
    exit(2);
  }

  structP->num = array_size;

  /*
   * Access data[] as if it had been allocated
   * as data[array_size].
   */
  for (unsigned int i = 0; i < array_size; ++i) {
    structP->data[i] = 1;
  }

  free(structP);
}
```

## Result Information

**Group:** Good Practice
**Language:**C (checker disabled if the analysis runs on C90 code indicated by the option `-c-version c90`)
**Default:** Off
**Command-Line Syntax:** `FLEXIBLE_ARRAY_MEMBER_INCORRECT_SIZE`
**Impact:** Low

# Version History

**Introduced in R2018b**

## See Also

`Hard-coded buffer size` | `Misuse of structure with flexible array member` | `Unprotected dynamic memory allocation` | `Pointer access out of bounds` | `Memory leak` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Incorrectly indented statement

Statement indentation incorrectly makes it appear as part of a block

## Description

This defect occurs when the indentation of a statement makes it appear as part of an `if`, `else` or another block but the arrangement or lack of braces actually keeps the statement outside the block.

### Risk

A developer or reviewer might incorrectly associate the statement with a block based on its indentation, leading to an incorrect assumption about the program logic.

For instance, in this example:

```
if(credentialsOK())
   login=1;
   setCookies();
```

the line `setCookies();` is not part of the `if` block, but the indentation suggests otherwise.

### Fix

If you want a statement to be part of a block, make sure that the statement is within the braces associated with the block. To identify the extent of a block, on the **Source** pane, click the opening brace.

If an `if`, `else` or `while` statement has no braces following the condition, only the next line on an execution path up to a semicolon is considered part of the `if`, `else` or `while` block. If you want subsequent lines to be included in the block, wrap the lines in braces.

For instance, in the preceding example, to include both statements in the `if` block, use:

```
if(credentialsOK()) {
   login=1;
   setCookies();
}
```

## Examples

### `else` Statement Incorrectly Indented

```
int switch1, switch2;

void doSomething(void);
void doSomethingElse(void);

void func() {
    if(switch1)
        if(switch2)
            doSomething();
    else
```

```
        doSomethingElse();
}
```

In this example, the `else` is indented as if it is associated with the first `if`. However, the `else` is actually associated with the second `if`. The indentation does not match the actual association and might lead to incorrect assumptions about the program logic.

#### Correction – Use Braces Appropriately

If you want the `else` to be associated with the first `if`, use braces to mark the boundaries of the first `if` block.

```
int switch1, switch2;

void doSomething(void);
void doSomethingElse(void);

void func() {
    if(switch1) {
        if(switch2)
            doSomething();
    }
    else
        doSomethingElse();
}
```

### Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `INCORRECT_INDENTATION`
**Impact:** Low

## Version History

**Introduced in R2020a**

### See Also

`Find defects (-checkers)` | `Semicolon on same line as if, for or while statement` | `Line with more than one statement`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Invalid scientific notation format

the use of an invalid format makes the code less readable

## Description

This issue occurs when you express a floating-point literal in exponential form but you do not use the standard scientific notation format of one nonzero digit before the decimal.

Polyspace does not flag the use of zero before the decimal to represent literal `0.0` in exponential form, for instance `0.00E+0`.

### Risk

The use of non-standard scientific notations to represent literals makes your code less readable and more error-prone.

### Fix

Use the standard format of one nonzero digit before the decimal to express floating-point literals in scientific notation.

## Examples

### Literals Represented with Invalid Scientific Format

```
#include <stdio.h>

void printUsingEFormat()
{
    float literalVals[] = {
        1154.12e+5L, //more than one digit before decimal
        .1E4F,       //no digit before decimal
        58.L,
        0.1E4F,     //zero digit before decimal
        1.34e+4,
        13.4e+3,    //more than one digit before decimal
        0.0000E+0,
        0xa.bp10l,
        0x15.0p5};
    for (int i = 0;
         i < sizeof(literalVals) / sizeof(literalVals[0]);
         ++i)
    {
        printf("%.6e\n", literalVals[i]);
    }
}
```

In this example, function `printUsingEFormat()` cycles through the elements of array `literalVals` and prints each element in scientific notation format.

Polyspace flags every literal that does not use a standard scientific notation format. The other array elements use the correct format or are not represented using scientific notation.

## Result Information

**Group:** Good Practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `INVALID_NOTATION_ON_E_CONSTANT`
**Impact:** Low

# Version History

**Introduced in R2022b**

## See Also

`Find defects (-checkers)`

# Macro terminated with a semicolon

Macro definition ends with a semicolon

## Description

This defect occurs when a macro that is invoked at least once has a definition ending with a semicolon.

### Risk

If a macro definition ends with a semicolon, the macro expansion can lead to unintended program logic in certain contexts, such as within an expression.

For instance, consider the macro:

```
#define INC_BY_ONE(x) ++x;
```

If used in the expression:

```
res = INC_BY_ONE(x)%2;
```

the expression resolves to:

```
res = ++x; %2;
```

The value of `x+1` is assigned to `res`, which is probably unintended. The leftover standalone statement `%2;` is valid C code and can only be detected by enabling strict compiler warnings.

### Fix

Do not end macro definitions with a semicolon. Leave it up to users of the macro to add a semicolon after the macro when needed.

Alternatively, use inline functions in preference to function-like macros that involve statements ending with semicolon.

## Examples

### Spurious Semicolon in Macro Definition

```
#define WHILE_LOOP(n) while(n>0);

void performAction(int timeStep);

void main() {
    int loopIter = 100;
    WHILE_LOOP(loopIter) {
        performAction(loopIter);
        loopIter--;
    }
}
```

In this example, the defect occurs because the definition of the macro `WHILE_LOOP(n)` ends with a semicolon. As a result of the semicolon, the `while` loop has an empty body and the subsequent statements in the block run only once. It was probably intended that the loop must iterate 100 times.

#### Correction – Remove Semicolon from Macro Definition

Remove the trailing semicolon from the macro definition. Users of the macro can add a semicolon after the macro when needed. In this example, a semicolon is not required.

```
#define WHILE_LOOP(n) while(n>0)

void performAction(int timeStep);

void main() {
    int loopIter = 100;
    WHILE_LOOP(loopIter) {
        performAction(loopIter);
        loopIter--;
    }
}
```

### Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SEMICOLON_TERMINATED_MACRO`
**Impact:** Low

## Version History

**Introduced in R2020a**

### See Also

`Find defects (-checkers)` | `Incorrectly indented statement` | `Semicolon on same line as if, for or while statement` | `Macro with multiple statements`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Line with more than one statement

Multiple statements on a line

## Description

This defect occurs when, before preprocessing starts, the analysis detects additional text after the semicolon (`;`) on a line. A defect is not raised for comments, `for`-loop definitions, braces, or backslashes.

### Risk

Use of one statement per line improves readability of the code. Since most statements in your code appear on a new line, use of multiple statements per line in a few cases within this arrangement can make code review difficult.

### Fix

Write one statement per line.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Examples

### Single-Line Initialization

```
int multi_init(void){
int abc = 4; int efg = 0; //defect

    return abc*efg;
}
```

In this example, `abc` and `efg` are initialized on the second line of the function as separate statements.

#### Correction — Comma-Separated Initialization

One possible correction is to use a comma instead of a semicolon to declare multiple variables on the same line.

```
int multi_init(void){
    int a = 4, b = 0;

    return a*b;
}
```

#### Correction — New Line for Each Initialization

One possible correction is to separate each initialization. By putting the initialization of b on the next line, the code longer raises a defect.

```
int multi_init(void){
    int a = 4;
    int b = 0;

    return a*b;
}
```

### Single-Line Loops

```
int multi_loop(void){
    int a, b = 0;
    int index = 1;
    int tab[9] = {1,1,2,3,5,8,13,21};

    for(a=0; a < 3; a++) {b+=a;} // no defect

for(b=0; b < 3; b++) {a+=b; index=b;} //defect

while (index < 7) {index++; tab[index] = index * index;} //defect
    return a*b;
}
```

In this example, there are three loops coded on single lines, each with multiple semicolons.

- The first for loop has multiple semicolons. Polyspace does not raise a defect for multiple statements within a for loop declaration.
- Polyspace does raise a defect on the second for loop because there are multiple statements after the for loop declaration.
- The while loop also has multiple statements after the loop declaration. Polyspace raises a defect on this line.

#### Correction — New Line for Each Loop Statement

One possible correction is to use a new line for each statement after the loop declaration.

```
int multi_loop(void){
    int a, b = 0;
    int index = 1;
    int tab[9] = {1,1,2,3,5,8,13,21};

    for(a=0; a < 3; a++) {b+=a;}

    for(b=0; b < 3; b++){
      a+=b;
      index=b;
    }

    while (index < 7){
      index++;
      tab[index] = index * index;
    }
    return a*b;
}
```

#### Single-line Conditionals

```
int multi_if(void){

    int a, b = 1;
    if(a == 0) { a++;} // no defect
else if(b == 1) {b++; a *= b;} //defect
}
```

In this example, there are two conditional statements an: `if` and an `else if`. The `if` line does not raise a defect because only one statement follows the condition. The `else if` statement does raise a defect because two statements follow the condition.

##### Correction — New Lines for Multi-Statement Conditionals

One possible correction is to use a new line for conditions with multiple statements.

```
int multi_if(void){
    int a, b = 1;

    if(a == 0) a++;
    else if(b == 1){
      b++;
      a *= b;
    }
}
```

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `MORE_THAN_ONE_STATEMENT`
**Impact:** Low

# Version History

**Introduced in R2013b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing break of switch case

case block of switch statement does not end in a break, [[fallthrough]] or explanatory comment

## Description

This defect occurs when a case block of switch statement does not end in a break, a [[fallthrough]], or a code comment.

If the last entry in the case block is a code comment, for instance:

```
switch (wt)
    {
      case WE_W:
        do_something_for_WE_W();
        do_something_else_for_WE_W();
        /* fall through to WE_X*/
      case WE_X:
        ...
    }
```

Polyspace assumes that the missing break is intentional and does not raise a defect.

### Risk

Switch cases without break statements fall through to the next switch case. If this fall-through is not intended, the switch case can unintentionally execute code and end the switch with unexpected results.

### Fix

If you forgot the break statement, add it before the end of the switch case.

If you do not want a break for the highlighted switch case, add a comment to your code to document why this case falls through to the next case. This comment removes the defect from your results and makes your code more maintainable.

In some cases, such as in template functions, the defect checker might ignore the comment at the end of a case statement. A cleaner way to indicate a deliberate fall through to the next case is to use one of these fallthrough attributes if possible:

- [[fallthrough]]: Available since C++17.
- [[gnu::fallthrough]] and [[clang::fallthrough]]: Available with GCC and Clang compilers.
- __attribute__((fallthrough)): Available with GCC compilers.

## Examples

### Switch Without Break Statements

```
enum WidgetEnum { WE_W, WE_X, WE_Y, WE_Z } widget_type;
```

```
extern void demo_do_something_for_WE_W(void);
extern void demo_do_something_for_WE_X(void);
extern void demo_report_error(void);

void bug_missingswitchbreak(enum WidgetEnum wt)
{
    /*
      In this non-compliant code example, the case where widget_type is WE_W lacks a
      break statement. Consequently, statements that should be executed only when
      widget_type is WE_X are executed even when widget_type is WE_W.
    */
    switch (wt)
    {
      case WE_W:
        demo_do_something_for_WE_W();
      case WE_X:
        demo_do_something_for_WE_X();
      default:
        /* Handle error condition */
        demo_report_error();
    }
}
```

In this example, there are two cases without `break` statements. When `wt` is `WE_W`, the statements for `WE_W`, `WE_X`, and the `default` case execute because the program falls through the two cases without a break. No defect is raised on the `default` case or last case because it does not need a break statement.

#### Correction — Add a Comment or break

To fix this example, either add a comment after the last statement in the case block and before the next case block to mark and document the acceptable fall-through or add a break statement to avoid fall-through. In this example, case `WE_W` is supposed to fall through, so a comment is added to explicitly state this action. For the second case, a break statement is added to avoid falling through to the `default` case.

```
enum WidgetEnum { WE_W, WE_X, WE_Y, WE_Z } widget_type;

extern void demo_do_something_for_WE_W(void);
extern void demo_do_something_for_WE_X(void);
extern void demo_report_error(void);

void corrected_missingswitchbreak(enum WidgetEnum wt)
{
    switch (wt)
    {
      case WE_W:
        demo_do_something_for_WE_W();
        /* fall through to WE_X*/
      case WE_X:
        demo_do_something_for_WE_X();
        break;
      default:
        /* Handle error condition */
        demo_report_error();
    }
}
```

## Result Information

**Group:** Good Practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `MISSING_SWITCH_BREAK`
**Impact:** Low

# Version History

**Introduced in R2016b**

## See Also

`Missing case for switch condition` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing overload of allocation or deallocation function

Only one function in an allocation-deallocation function pair is overloaded

## Description

This defect occurs when you overload `operator new` but do not overload the corresponding `operator delete`, or vice versa.

### Risk

You typically overload `operator new` to perform some bookkeeping in addition to allocating memory on the free store. Unless you overload the corresponding `operator delete`, it is likely that you omitted some corresponding bookkeeping when deallocating the memory.

The defect can also indicate a coding error. For instance, you overloaded the placement form of `operator new[]`:

```
void *operator new[](std::size_t count, void *ptr);
```

but the non-placement form of `operator delete[]`:

```
void operator delete[](void *ptr);
```

instead of the placement form:

```
void operator delete[](void *ptr, void *p );
```

### Fix

When overloading `operator new`, make sure that you overload the corresponding `operator delete` in the same scope, and vice versa.

For instance, in a class, if you overload the placement form of `operator new`:

```
class MyClass {
   void* operator new  ( std::size_t count, void* ptr ){
   ...
   }
};
```

Make sure that you also overload the placement form of `operator delete`:

```
class MyClass {
   void operator delete  ( void* ptr, void* place ){
   ...
   }
};
```

To find the `operator delete` corresponding to an `operator new`, see the reference pages for `operator new` and `operator delete`.

## Examples

### Mismatch Between Overloaded operator new and operator delete

```
#include <new>
#include <cstdlib>

int global_store;

void update_bookkeeping(void *allocated_ptr, bool alloc) {
   if(alloc)
      global_store++;
   else
      global_store--;
}


void *operator new(std::size_t size, const std::nothrow_t& tag);
void *operator new(std::size_t size, const std::nothrow_t& tag) {
    void *ptr = (void*)malloc(size);
    if (ptr != nullptr)
        update_bookkeeping(ptr, true);
    return ptr;
}

void operator delete[](void *ptr, const std::nothrow_t& tag);
void operator delete[](void* ptr, const std::nothrow_t& tag) {
    update_bookkeeping(ptr, false);
    free(ptr);
}
```

In this example, the operators `operator new` and `operator delete[]` are overloaded but there are no overloads of the corresponding `operator delete` and `operator new[]` operators.

The overload of `operator new` calls a function `update_bookkeeping` to change the value of a global variable `global_store`. If the default `operator delete` is called, this global variable is unaffected, which might defy developer's expectations.

#### Correction — Overload the Correct Form of operator delete

If you want to overload `operator new`, overload the corresponding form of `operator delete` in the same scope.

```
#include <new>
#include <cstdlib>

int global_store;

void update_bookkeeping(void *allocated_ptr, bool alloc) {
   if(alloc)
      global_store++;
   else
      global_store--;
}
```

```
void *operator new(std::size_t size, const std::nothrow_t& tag);
void *operator new(std::size_t size, const std::nothrow_t& tag) {
    void *ptr = (void*)malloc(size);
    if (ptr != nullptr)
        update_bookkeeping(ptr, true);
    return ptr;
}

void operator delete(void *ptr, const std::nothrow_t& tag);
void operator delete(void* ptr, const std::nothrow_t& tag) {
    update_bookkeeping(ptr, false);
    free(ptr);
}
```

### Result Information

**Group:** Good practice
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `MISSING_OVERLOAD_NEW_DELETE_PAIR`
**Impact:** Low

## Version History

**Introduced in R2019a**

### See Also

`Invalid free of pointer` | `Invalid deletion of pointer` | `Memory leak` | `Mismatched alloc/dealloc functions on Windows` | `Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Missing reset of freed pointer

Pointer `free` not followed by a reset statement to clear leftover data

## Description

This defect occurs when a pointer is freed and not reassigned another value. After freeing a pointer, the memory data is still accessible. To clear this data, the pointer must also be set to NULL or another value.

### Risk

Not resetting pointers can cause dangling pointers. Dangling pointers can cause:

- Freeing already freed memory.
- Reading from or writing to already freed memory.
- Hackers executing code stored in freed pointers or with vulnerable permissions.

### Fix

After freeing a pointer, if it is not immediately assigned to another valid address, set the pointer to NULL.

## Examples

### Free Without Reset

```
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};

void allocateAndFreeMemory()
{
    static char *str = NULL;

    if (str == NULL)
        str = (char *)malloc(SIZE20);

    if (str != NULL)
        free(str);
}
```

In this example, the pointer `str` is freed at the end of the program. The next call to `allocateAndFreeMemory` can fail because `str` is not NULL and the initialization to NULL can be invalid.

#### Correction — Redefine `free` to Free and Reset

One possible correction is to customize `free` so that when you free a pointer, it is automatically reset.

```
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};

static void sanitize_free(void **p)
{
    if ((p != NULL) && (*p != NULL))
    {
        free(*p);
        *p = NULL;
    }
}

#define free(X) sanitize_free((void **)&X)

void allocateAndFreeMemory()
{
    static char *str = NULL;

    if (str == NULL)
        str = (char *)malloc(SIZE20);

    if (str != ((void *)0))
    {
        free(str);
    }
}
```

### Result Information

**Group:** Good Practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `MISSING_FREED_PTR_RESET`
**Impact:** Low

## Version History

**Introduced in R2016b**

### See Also

`Use of previously freed pointer` | `Invalid free of pointer` | `Find defects (-checkers)`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Macro with multiple statements

Macro consists of multiple semicolon-terminated statements, enclosed in braces or not

## Description

This defect occurs when a macro contains multiple semicolon-terminated statements, irrespective of whether the statements are enclosed in braces.

### Risk

The macro expansion, in certain contexts such as an `if` condition or a loop, can lead to unintended program logic.

For instance, consider the macro:

```
#define RESET(x,y) \
   x=0; \
   y=0;
```

In an `if` statement such as:

```
if(checkSomeCondition)
   RESET(x,y);
```

the macro expands to:

```
if(checkSomething)
   x=0;
y=0;
```

which might be unexpected if you want both statements to be executed in an `if` block.

### Fix

In a macro definition, wrap multiple statements in a `do...while(0)` loop.

For instance, in the preceding example, use the definition:

```
#define RESET(x,y) \
   do { \
     x=0; \
     y=0; \
   } while(0)
```

This macro is appropriate to expand in all contexts. The `while(0)` ensures that the statements are executed only once.

Alternatively, use inline functions in preference to function-like macros that involve multiple statements.

Note that the loop is required for the correct solution and wrapping the statements in braces alone does not fix the issue. The macro expansion can still lead to unintended code.

## Examples

### Macro with Multiple Statements

```
#define RESET(x,y) \
   x=0; \
   y=0;

void func(int *x, int *y, int resetFlag){
    if(resetFlag)
        RESET(x,y);
}
```

In this example, the defect occurs because the macro RESET consists of multiple statements.

#### Correction – Wrap Multiple Statements of Macro in do-while Loop

Wrap the statements of the macro in a do..while(0) loop in the macro definition.

```
#define RESET(x,y) \
   do { \
     x=0; \
     y=0; \
   } while(0)

void func(int *x, int *y, int resetFlag){
    if(resetFlag)
        RESET(x,y);
}
```

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** MULTI_STMT_MACRO
**Impact:** Low

# Version History

**Introduced in R2020a**

## See Also

Find defects (-checkers) | Macro terminated with a semicolon | Incorrectly indented statement | Semicolon on same line as if, for or while statement

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Possibly inappropriate data type for switch expression

switch expression has a data type other than char, short, int or enum

## Description

This defect occurs when a switch expression has a data type other than char, short, int or enum.

The checker flags other integer data types such as boolean types, bit fields, or long.

### Risk

It is preferred to use char, short, int or enum in switch expressions instead of:

- Boolean types, because a switch expression with a boolean type can be replaced with an if condition that evaluates the same expression. A switch expression is too heavy for a simple control flow based on a boolean condition.
- Bit field types, because bit field types imply memory restrictions. If you just want to specify a variable with a finite number of values, enumerations are preferred since they enable a more readable code.
- Types with size greater than int because a switch expression that requires a type with size greater than int implies too many case labels and can be possibly redesigned.

Non-integer types are not supported in switch expressions.

### Fix

Use variables of char, short, int or enum data types in switch expressions.

## Examples

### Use of Inappropriate Types in switch Expressions

```
#ifndef __cplusplus
#include <stdbool.h>
#endif

void func(bool s) {
    switch(s) {
        case 0: //Perform some operation
        break;
        case 1: //Perform another operation
        break;
    }
}
```

In this C++ example, the checker flags the use of a bool variable in a switch expression.

#### Correction – Use `if` Condition Instead of `switch`

If the `switch` expression indeed requires two values, use an `if` statement instead.

```
#ifndef __cplusplus
#include <stdbool.h>
#endif


void func(bool s) {
    if(s) {
        //Perform some operation
        }
    else {
        //Perform another operation
        }
}
```

#### Correction – Use Different Data Type

If you anticipate adding more labels to the `switch` expression later, use a data type that can accommodate larger values.

```
void func(char s) {
    switch(s) {
        case 0: //Perform some operation
        break;
        case 1: //Perform another operation
        break;
        default: //Default behavior
    }
}
```

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `INAPPROPRIATE_TYPE_IN_SWITCH`
**Impact:** Low

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)`

#### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Semicolon on the same line as an if, for or while statement

Semicolon on same line results in empty body of `if`, `for` or `while` statement

## Description

This defect occurs when a semicolon on the same line as the last token of an `if`, `for` or `while` statement results in an empty body.

The checker makes an exception for the case where the `if` statement is immediately followed by an `else` statement:

```
if(condition);
else {
  ...
}
```

### Risk

The semicolon following the if, for or while statement often indicates a programming error. The spurious semicolon changes the execution flow and leads to unintended results.

### Fix

If you want an empty body for the `if`, `for` or `while` statement, wrap the semicolon in a block and place the block on a new line to explicitly indicate your intent:

```
if(condition)
   {;}
```

Otherwise, remove the spurious semicolon.

## Examples

### Spurious Semicolon

```
int credentialsOK(void);

void login () {
    int loggedIn = 0;
    if(credentialsOK());
      loggedIn = 1;
}
```

In this example, the spurious semicolon results in an empty `if` body. The assignment `loggedIn=1` is always performed. However, the assignment was probably to be performed only under a condition.

#### Correction – Remove Spurious Semicolon

If the semicolon was unintended, remove the semicolon.

```
int credentialsOK(void);

void login () {
    int loggedIn = 0;
    if(credentialsOK())
      loggedIn = 1;
}
```

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SEMICOLON_CTRL_STMT_SAME_LINE`
**Impact:** Low

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)` | `Macro terminated with a semicolon` | `Incorrectly indented statement` | `Macro with multiple statements`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unmodified variable not const-qualified

Variable not `const`-qualified but variable value not modified during lifetime

## Description

This defect occurs when a local variable is not `const`-qualified and one of the following statements is true during the variable lifetime:

- You do not perform write operations on the variable after initialization.
- When you perform write operations, you reassign the same constant value to the variable.

The checker considers a variable as modified if its address is assigned to a pointer or reference (unless it is a pointer or reference to a `const` variable), passed to another function, or otherwise used. In these situations, the checker does not suggest adding a `const` qualifier.

The checker flags arrays as candidates for `const`-qualification only if you do not perform write operations on the array elements at all after initialization.

### Risk

`const`-qualifying a variable avoids unintended modification of the variable during later code maintenance. The `const` qualifier also indicates to a developer that the variable retains its initial value in the remainder of the code.

### Fix

If you do not expect to modify a variable value during its lifetime, add the `const` qualifier to the variable declaration and initialize the variable at declaration.

If you expect the variable to be modified, see if the absence of a modification indicates a programming omission and fix the issue.

## Examples

### Missing `const` Qualification on Pointer

```
#include <string.h>

char returnNthCharacter (int n) {
    char* pwd = "aXeWdf10fg" ;
    char nthCharacter;

    for(int i=0; i < strlen(pwd); i++) {
        if(i==n)
            nthCharacter = pwd[i];
    }
    return nthCharacter;
}
```

In this example, the pointer pwd is not `const`-qualified. However, beyond initialization with a constant, it is not reassigned anywhere in the `returnNthCharacter` function.

#### Correction – Add const at Variable Declaration

If the variable is not intended to be modified, add the const qualifier at declaration. In this example, both the pointer and the pointed variable are not modified. Add a const qualifier to both the pointer and the pointed variable. Later modifications cannot reassign the pointer pwd to point at a different variable nor modify the value at the pointed location.

```
#include <string.h>

char returnNthCharacter (int n) {
    const char* const pwd = "aXeWdf10fg" ;
    char nthCharacter;

    for(int i=0; i < strlen(pwd); i++) {
        if(i==n)
            nthCharacter = pwd[i];
    }
    return nthCharacter;
}
```

Note that the checker only flags the missing const from the pointer declaration. The checker does not determine if the pointed location also merits a const qualifier.

### Reassignment of Variable to Initial Value

```
void resetBuffer(int aCondition) {
    int addr = 0xff;
    if(aCondition){
        addr = 0xff;
    }
    else {
        addr = 0xff;
    }
}
```

In this example, the variable addr is initialized to a value and reassigned the same value twice. In larger code examples, such issues can easily arise from copy-paste errors.

#### Correction – Fix Programming Error

The reassignment in this example indicates a possible programming error. One possible correction is to fix the programming error and thereby avoid reassigning the same value.

```
void resetBuffer(int aCondition) {
    int addr = 0xff;
    if(aCondition){
        addr = 0x00;
    }
}
```

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** UNMODIFIED_VAR_NOT_CONST
**Impact:** Low

## Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)` | `Expensive local variable copy`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Unused parameter

Function prototype has parameters not read or written in function body

## Description

This defect occurs when a function parameter is neither read nor written in the function body. The checker does not flag unused parameters in functions with empty bodies.

### Risk

Unused parameters can indicate that the code is possibly incomplete. The parameter is possibly intended for an operation that you forgot to code.

If the copied objects are large, redundant copies can slow down performance.

### Fix

Determine if you intend to use the parameters. Otherwise, remove parameters that you do not use in the function body.

You can intentionally have unused parameters. For instance, you have parameters that you intend to use later when you add enhancements to the function. Add a code comment indicating your intention for later use. The code comment helps you or a code reviewer understand why your function has unused parameters.

Alternatively, add a statement such as `(void)var;` in the function body. `var` is the unused parameter. You can define a macro that expands to this statement and add the macro to the function body.

## Examples

### Unused Parameter

```
void func(int* xptr, int* yptr, int flag) {
    if(flag==1) {
        *xptr=0;
    }
    else {
        *xptr=1;
    }
}

int main() {
    int x,y;
    func(&x,&y,1);
    return 0;
}
```

In this example, the parameter `yptr` is not used in the body of `func`.

### Correction — Use Parameter

One possible correction is to check if you intended to use the parameter. Fix your code if you intended to use the parameter.

```
void func(int* xptr, int* yptr, int flag) {
    if(flag==1) {
        *xptr=0;
        *yptr=1;
    }
    else {
        *xptr=1;
        *yptr=0;
    }
}

int main() {
    int x,y;
    func(&x,&y,1);
    return 0;
}
```

### Correction — Explicitly Indicate Unused Parameter

Another possible correction is to explicitly indicate that you are aware of the unused parameter.

```
#define UNUSED(x) (void)x

void func(int* xptr, int* yptr, int flag) {
    UNUSED(yptr);
    if(flag==1) {
        *xptr=0;
    }
    else {
        *xptr=1;
    }
}

int main() {
    int x,y;
    func(&x,&y,1);
    return 0;
}
```

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `UNUSED_PARAMETER`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Useless include

An include directive is present but not used

## Description

This defect occurs when you do not use an included header.

For instance, when you do not:

- Invoke macros defined in the included file or the included file's includes.
- Use data types defined in the included file or the included file's includes.
- Call or use references to an object or function defined in the included file or the included file's includes.
- Use an externally visible object provided by the included file outside of the file.

### Risk

Unnecessary file dependencies slow the build speed of the project. To improve build speed, remove include directives that are not used.

### Fix

Remove the unused include directive.

## Examples

### Unused Include Directive

This example does not use the `<cmath>` and `<stdexcedpt>` header files or their include directives.

```
#include <iostream>     //Compliant
#include <cmath>        //Noncompliant
#include <stdexcept>    //Noncompliant
#include <vector>       //Compliant

int age = 0;
std::vector<int> allAges;

void saveAge()
{
    std::cout << "Enter age: ";
    std::cin >> age;

    allAges.push_back(age);
}

int main() {
    while (age <= 40)
    {
        saveAge();
    }

    for (int x : allAges)
    {
        std::cout << x << std::endl;
    }

    return 0;
}
```

**Correction — Remove Unused Include Directives**

One correction is to delete the unused include directives from the file.

```
#include <iostream>
#include <vector>

int age = 0;
std::vector<int> allAges;

void saveAge()
{
    std::cout << "Enter age: ";
    std::cin >> age;

    allAges.push_back(age);
}

int main() {
    while (age <= 40)
    {
        saveAge();
    }

    for (int x : allAges)
    {
        std::cout << x << std::endl;
    }

    return 0;
}
```

## Result Information

**Group:** Good Practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `USELESS_INCLUDE`
**Impact:** Low

# Version History

**Introduced in R2022b**

## See Also

`Find defects (-checkers)`

# Use of a forbidden function

Use of function that appears in a blocklist of forbidden functions

## Description

This defect occurs when you use a function that appears in a blocklist of forbidden functions. To create the blocklist:

- List functions in an XML file in a specific syntax.

  Copy the template file `code-behavior-specifications-template.xml` from the folder *`polyspaceroot`*`\polyspace\verifier\cxx` to a writable location and modify the file. Enter each function in the file using the following syntax after existing similar entries:

  ```
  <function name="funcname">
     <behavior name="FORBIDDEN_FUNC"/>
  </function>
  ```

  where *`funcname`* is the name of the function you want to block.
- Specify this XML file as argument for the option `-code-behavior-specifications`. See “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers”.

### Risk

A function might be blocked for one of these reasons:

- The function can lead to many situations where the behavior is undefined leading to security vulnerabilities, and a more secure function exists.

  You can forbid functions that are not explicitly checked by existing checkers such as `Use of dangerous standard function` or `Use of obsolete standard function`.
- The function is being deprecated as part of a migration, for instance, from C++98 to C++11.

  As part of a migration, you can make a list of functions that need to be replaced and use this checker to identify their use.

### Fix

Replace the blocked function with an allowed function.

When rolling out this checker to a group, project or organization, create a list of blocked functions and their replacements so that results reviewers can consult the list and make appropriate replacements.

### Extend Checker

This defect checker requires a blocklist of functions to be specified. Even if you specify the checker using the option `Find defects (-checkers)`, it is not enabled unless you also specify the blocklist. See “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers”.

## Examples

### Use of Forbidden Function `std::signal`

```
#include <csignal>
#include <iostream>

namespace
{
  volatile std::sig_atomic_t gSignalStatus;
}

void signal_handler(int signal)
{
  gSignalStatus = signal;
}

int main()
{
  // Install a signal handler
  std::signal(SIGINT, signal_handler);

  std::cout << "SignalValue: " << gSignalStatus << '\n';
  std::cout << "Sending signal " << SIGINT << '\n';
  std::raise(SIGINT);
  std::cout << "SignalValue: " << gSignalStatus << '\n';
}
```

Suppose you want to deprecate the `std::signal` function. Add the following to the template XML file after similar existing entries:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <functions>
     <function name="std::signal">
        <behavior name="FORBIDDEN_FUNC"/>
     </function>
   </functions>
</specifications>
```

and specify the XML file with the option `-code-behavior-specifications`.

In the analysis results, all uses of the `std::signal` function are flagged by this checker.

### Blocking C++ Overloaded Operators

```
class orderedPair {
        int var1;
        int var2;
    public:
        orderedPair() {
            var1 = 0;
            var2 = 0;
        }
        orderedPair(int arg1, int arg2) {
            var1 = arg1;
            var2 = arg2;
        }
        orderedPair& operator=(const orderedPair& rhs) {
            var1 = rhs.var1;
            var2 = rhs.var2;
            return *this;
        }
        orderedPair& operator+(orderedPair& rhs) {
            var1 += rhs.var1;
            var2 += rhs.var2;
            return *this;
        }
};

void main() {
    int one=1, zero=0, sum;
    orderedPair firstOrderedPair(one, one);
    orderedPair secondOrderedPair(zero, one);
    orderedPair sumPair;

    sum = zero + one;
    sumPair = firstOrderedPair + secondOrderedPair;
}
```

Suppose you want to identify all the locations where operator overloads in the `orderedPair` class are used. Add the overloaded operators to the template XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <functions>
     <function name="orderedPair::operator=">
        <behavior name="FORBIDDEN_FUNC"/>
     </function>
     <function name="orderedPair::operator+">
        <behavior name="FORBIDDEN_FUNC"/>
     </function>
   </functions>
</specifications>
```

and specify the XML file with the option `-code-behavior-specifications`.

The analysis identifies all calls to the overloaded operators and flags their use. Using this method, you can distinguish specific overloads of an operator instead of searching for and browsing through all instances of the operator.

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `FORBIDDEN_FUNC`
**Impact:** Low

# Version History

**Introduced in R2020a**

## See Also

`Find defects (-checkers)` | `-code-behavior-specifications`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers"

# Use of a forbidden C/C++ keyword

Use of keyword that appears in a blocklist of forbidden keywords

## Description

This defect occurs when you use a keyword that appears in a blocklist of forbidden keywords. To create the blocklist:

- List keywords in an XML file in a specific syntax.

  Copy the template file `code-behavior-specifications-template.xml` from the folder *`polyspaceroot`*`\polyspace\verifier\cxx` to a writable location and modify the file. Enter each keyword in the file using the following syntax after existing similar entries:

  ```
  <token name="keywordname" kind="keyword">
        <behavior name="FORBIDDEN_KEYWORD"/>
  </token>
  ```

  where *`keywordname`* is the name of the keyword you want to block.
- Specify this XML file as argument for the option `-code-behavior-specifications`. See “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers”.

The checker can be used to forbid keywords with whitespaces such as `for each` or `enum class`.

### Risk

A keyword might be blocked for one of these reasons:

- The keyword is prone to misuse or makes the code difficult to maintain.
- The keyword is being deprecated as part of a migration, for instance, from C++98 to C++11.

  As part of a migration, you can make a list of keywords that need to be replaced and use this checker to identify their use.

### Fix

Replace the blocked keyword with an allowed keyword.

When rolling out this checker to a group, project or organization, create a list of blocked keywords and their replacements so that results reviewers can consult the list and make appropriate replacements.

### Extend Checker

This defect checker requires a blocklist of keywords to be specified. Even if you specify the checker using the option `Find defects (-checkers)`, it is not enabled unless you also specify the blocklist. See “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers”.

## Examples

### Use of Forbidden Keyword `final`

```
class Base {
public:
    virtual bool isThisFinal() const {
        return false;
    }
};
class final : public Base {
    virtual bool isThisFinal() const
    override
    final {
        return true;
    }
};
```

Suppose you want to deprecate the `final` keyword. Add the following to the template XML file after similar existing entries:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <tokens>
    <token name="final" kind="keyword">
       <behavior name="FORBIDDEN_KEYWORD"/>
    </token>
  </tokens>
</specifications>
```

and specify the XML file with the option `-code-behavior-specifications`.

In the analysis results, all uses of the `final` keyword are flagged by this checker. Note that the checker flags only the keyword `final` and not the class name `final`.

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `FORBIDDEN_KEYWORD`
**Impact:** Low

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)` | `-code-behavior-specifications`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"

"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers"

# Use of a forbidden macro

Use of macro that appears in a blocklist of forbidden macros

## Description

This defect occurs when you use a macro that appears in a blocklist of forbidden macros. To create the blocklist:

- List macros in an XML file in a specific syntax.

  Copy the template file `code-behavior-specifications-template.xml` from the folder *`polyspaceroot`*`\polyspace\verifier\cxx` to a writable location and modify the file. Enter each macro in the file using the following syntax after existing similar entries:

  ```
  <token name="macroname" kind="macro">
        <behavior name="FORBIDDEN_MACRO"/>
  </token>
  ```

  where *`macroname`* is the name of the macro you want to block.
- Specify this XML file as argument for the option `-code-behavior-specifications`. See “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers”.

### Risk

A macro might be blocked for one of these reasons:

- The macro is prone to misuse or makes the code difficult to maintain.
- The macro is being deprecated as part of a migration. For instance, you might want to allow the boolean macros `TRUE` and `FALSE` only in their all-caps form, and forbid other forms such as `true` and `false`.

  As part of a migration, you can make a list of macros that need to be replaced and use this checker to identify their use.

### Fix

Replace the blocked macro with an allowed macro.

When rolling out this checker to a group, project or organization, create a list of blocked macros and their replacements so that results reviewers can consult the list and make appropriate replacements.

### Extend Checker

This defect checker requires a blocklist of macros to be specified. Even if you specify the checker using the option `Find defects (-checkers)`, it is not enabled unless you also specify the blocklist. See “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers”.

## Examples

### Use of Forbidden Macros `true` and `false`

```
#define TRUE 1u
#define FALSE 0u
#define true 1u
#define false 0u

unsigned char isPositive(int num){
    if(num > 0)
        return TRUE;
    else
        return FALSE;
}

unsigned char isEven(int num){
    if(num%2 == 0)
        return true;
    else
        return false;
}
```

Suppose you want to allow the boolean macros `TRUE` and `FALSE` only in their all-caps form, and forbid other forms such as `true` and `false`. Add the following to the template XML file after similar existing entries:

```
<?xml version="1.0" encoding="UTF-8"?>
<specifications xmlns="http://www.mathworks.com/PolyspaceCodeBehaviorSpecifications">
  <tokens>
    <token name="true" kind="macro">
       <behavior name="FORBIDDEN_MACRO"/>
    </token>
    <token name="false" kind="macro">
       <behavior name="FORBIDDEN_MACRO"/>
    </token>
  </tokens>
</specifications>
```

and specify the XML file with the option `-code-behavior-specifications`.

In the analysis results, all uses of the `true` and `false` macros are flagged by this checker.

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `FORBIDDEN_MACRO`
**Impact:** Low

# Version History

**Introduced in R2022b**

## See Also

`Find defects (-checkers)` | `-code-behavior-specifications`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"
"Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers"

# Use of setjmp/longjmp

`setjmp` and `longjmp` cause deviation from normal control flow

## Description

This defect occurs when you use a combination of `setjmp` and `longjmp` or `sigsetjmp` and `siglongjmp` to deviate from normal control flow and perform non-local jumps in your code.

### Risk

Using `setjmp` and `longjmp`, or `sigsetjmp` and `siglongjmp` has the following risks:

- Nonlocal jumps are vulnerable to attacks that exploit common errors such as buffer overflows. Attackers can redirect the control flow and potentially execute arbitrary code.
- Resources such as dynamically allocated memory and open files might not be closed, causing resource leaks.
- If you use `setjmp` and `longjmp` in combination with a signal handler, unexpected control flow can occur. POSIX does not specify whether `setjmp` saves the signal mask.
- Using `setjmp` and `longjmp` or `sigsetjmp` and `siglongjmp` makes your program difficult to understand and maintain.

### Fix

Perform nonlocal jumps in your code using `setjmp/longjmp` or `sigsetjmp/siglongjmp` only in contexts where such jumps can be performed securely. Alternatively, use POSIX threads if possible.

In C++, to simulate throwing and catching exceptions, use standard idioms such as `throw` expressions and `catch` statements.

## Examples

### Use of `setjmp` and `longjmp`

```
#include <setjmp.h>
#include <signal.h>

extern int update(int);
extern void print_int(int);

static jmp_buf env;
void sighandler(int signum) {
    longjmp(env, signum);
}
void func_main(int i) {
    signal(SIGINT, sighandler);
    if (setjmp(env)==0) {
        while(1) {
            /* Main loop of program, iterates until SIGINT signal catch */
            i = update(i);
        }
    } else {
        /* Managing longjmp return */
        i = -update(i);
    }

    print_int(i);
    return;
}
```

In this example, the initial return value of `setjmp` is 0. The `update` function is called in an infinite `while` loop until the user interrupts it through a signal.

In the signal handling function, the `longjmp` statement causes a jump back to `main` and the return value of `setjmp` is now 1. Therefore, the `else` branch is executed.

**Correction — Use Alternative to `setjmp` and `longjmp`**

To emulate the same behavior more securely, use a `volatile` global variable instead of a combination of `setjmp` and `longjmp`.

```
#include <setjmp.h>
#include <signal.h>

extern int update(int);
extern void print_int(int);

volatile sig_atomic_t eflag = 0;

void sighandler(int signum) {
     eflag = signum;                   /* Fix: using global variable */
}

void func_main(int i) {
      /* Fix: Better design to avoid use of setjmp/longjmp */
    signal(SIGINT, sighandler);
    while(!eflag) {                   /* Fix: using global variable */
        /* Main loop of program, iterates until eflag is changed */
        i = update(i);
    }

    print_int(i);
    return;
}
```

## Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SETJMP_LONGJMP_USE`
**Impact:** Low

# Version History

**Introduced in R2015b**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

### External Websites

Linux man page for setjmp

# Redundant expression in sizeof operand

`sizeof` operand contains expression that is not evaluated

## Description

This defect occurs when a `sizeof` operand contains expressions whose evaluation does not affect the `sizeof` result. In place of the current expression in the `sizeof` operand, a data type, a variable or a simpler expression could have been used without any loss of functionality.

### Risk

In situations flagged by this defect, the expression in the `sizeof` operand is needlessly complicated, reduces the code readability and adds to maintainability costs. The expression might also give a false impression about the result of the `sizeof` operand.

For instance, consider the expression:

```
sizeof(void (*[n])(int arr[U+V]))
```

The operand of `sizeof` is an array of `n` function pointers, each of type `void () (int*)`. The additional `U+V`, which is not evaluated, makes the full expression needlessly complicated. The expression also gives the false impression that the function pointer argument being an array of size `U+V` matters for the `sizeof` result.

### Fix

The first event in the defect traceback shows where the redundant subexpression of the `sizeof` operand begins.

Simplify or completely remove the redundant expression. When possible, use a data type as the `sizeof` operand. For instance, in the preceding example, a simpler equivalent `sizeof` operation is:

```
sizeof(void (*[n])(int*))
```

If you want the expression to be evaluated, perform the evaluation in a separate statement.

## Examples

### Unnecessarily Complex Expression in `sizeof` Operand

```
void func() {
    int size1, size2, size3;
    char x = 0;
    short y = 0;
    int z = 0, w = 0;

    size1 = sizeof(x + y);
    size2 = sizeof(x + z);
    size3 = sizeof(z + w);
}
```

In this example, the defect checker flags the second and third `sizeof` operation because the expressions in the `sizeof` operand can be simplified without changing the `sizeof` result.

The checker does not flag the first operation because the expression in the `sizeof` operand cannot be simplified further without affecting the `sizeof` result.

#### Correction – Simplify Expression in `sizeof` Operand

Simplify the expression in the `sizeof` operand. In the following corrections, the `sizeof` results are the same as with the preceding expressions.

```
void func() {
    int size1, size2, size3;
    char x = 0;
    short y = 0;
    int z = 0, w = 0;

    size1 = sizeof(x + y);
    size2 = sizeof(z);
    size3 = sizeof(z);
}
```

### Result Information

**Group:** Good practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `SIZEOF_USELESS_OP`
**Impact:** Low

## Version History

**Introduced in R2020a**

### See Also

`Find defects (-checkers)` | `Side effect of expression ignored`

#### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# File does not compile

File has a compilation error

## Description

This defect occurs when Polyspace cannot analyze a file because of compilation errors. The defect is located on the first line and column of the file, and indicates that the file has one or more compilation errors.

To find the actual compilation errors, see the analysis log. For information on how to investigate further in:

- Polyspace Bug Finder or Polyspace Bug Finder Server, see "View Error Information When Analysis Stops".
- Polyspace as You Code, see how to follow the progress of analysis in your IDE. See steps in Visual Studio, Visual Studio Code or Eclipse.

Use this checker to find out at a glance whether you have files with compilation issues in an integration analysis (Polyspace Bug Finder or Polyspace Bug Finder Server) or whether the current file being analyzed does not compile yet (Polyspace as You Code). Using this checker saves you from opening the analysis log each time to find out if there are compilation issues. You can determine from your analysis results if a file did not compile.

### Risk

Typically, your compiler can also find the issues that this defect checker finds.

If your file compiles with your compiler but the compilation phase of a Polyspace analysis fails, it means that your analysis configuration does not emulate your compiler accurately. For instance, if the analysis fails because a standard library function appears to be undefined, you might have to explicitly specify the folders containing your compiler headers, use the `polyspace-configure` command to extract the paths, or otherwise improve your configuration.

### Fix

Identify all compilation errors from the analysis log and fix them.

## Examples

### Undefined Identifier

```
int nthFib (int n) {
    int i=0, sum=1;
    for (int iter = 0; iter < n; iter++) {
      int t = i;
      i = sum;
      sum += t;
    }
    return iter;
}
```

In this example, the variable `iter` is defined only in the `for` loop. But the `return` statement outside the loop refers to the variable, causing an undefined identifier error. (The compilation error here also indicates a logical error since the intent was to return the variable `sum`.)

**(C++ Only) Missing Header**

```
#include <cstdio>
#include <cstdlib>

void func() {
   char* message = (char*)malloc (strlen ("Hello, World\n")+1);
   strcpy (message, "Hello, World\n");
   printf ("%s", message);
   free (message);
}
```

In this example, the string functions `strlen` and `strcpy` are used but the header file `cstring` is not included. This leads to the functions appearing as undefined.

## Result Information

**Group:** Good Practice
**Language:** C | C++
**Default:** Off
**Command-Line Syntax:** `FILE_DOES_NOT_COMPILE`
**Impact:** Low

# Version History

**Introduced in R2021a**

## See Also

`Find defects (-checkers)` | `Stop analysis if a file does not compile (-stop-if-compile-error)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Declaration of catch for generic exception

A `catch` block handles a generic exception that might have many different subtypes

## Description

This defect occurs when you design `catch` blocks to handle generic high level exceptions. Polyspace checks the exception specifications of a function and raises a violation if a catch block handles:

- All exceptions by using a catch-all block.
- Exception of the class `std::exception`. Because `std::exception` is the base class for all standard C++ exceptions, a catch block that handles `std::exception` type matches all derived exceptions.

### Risk

Using generic catch blocks hides the emergence of unexpected new exceptions and hinders their proper handling. Such generic catch blocks makes the code vulnerable to security issues. For instance:

```
void foo(void){
    try{
        //..
    }catch(std::exception& e){
        //Log error
    }
}
```

Because the `catch` block handles the generic `std::exception` class, this code hides unexpected exceptions or those that need to be handled differently. For instance, If an unexpected `std::invalid_argument` exception is raised in `foo()`, it might remain undetected by the developer because it is matched with the catch block. Because the catch block is not programmed to handle `std::invalid_argument` properly, the poorly handled exception becomes a vulnerability for the code.

### Fix

To fix this defect, avoid catching high-level generic exceptions. Write catch blocks that handle specific exceptions to enable handling different exceptions in different ways. Unexpected or new exceptions are also easily detected when catch blocks are specific.

## Examples

### Avoid Generic Catch Blocks

```
#include<string>
#include<exception>
#include<stdexcept>

extern void logger(std::string s, std::exception& e);
void foo(void){
    try{
        //..
    }catch(std::exception& e){
         logger("foo failed", e);
    }
}
```

In this example, Polyspace flags the use of the generic `catch` block.

**Correction — Use Specific Catch Blocks**

To resolve this defect, use specific catch blocks. When you use specific catch blocks, new and unexpected exceptions become unhandled exceptions that are easily detected.

```
#include<string>
#include<exception>
extern void logger(std::string s, std::exception& e);
void foo(void){
    try{
        //..
    }catch(std::domain_error& e){
         logger("Domain Error in foo", e);
    }catch(std::invalid_argument& e){
        logger("Invalid Argument in foo", e);
    }
}
```

## Result Information

**Group:** Good practice
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `CATCH_FOR_GENERIC_EXCEPTION`
**Impact:** Low

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

**Topics**

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Declaration of throw for generic exception

A function throws a generic exception, which might have many subtypes

## Description

This defect occurs when a function throws a generic exception. Polyspace checks the throw statements, dynamic exception specifications, or the documentation of a function and raises this defect if any of these conditions is true:

- The function contains `throw` statements that raise generic exceptions.
- The dynamic exception specification contains generic exceptions.
- The function is documented with a `@throw` comment that contains generic exceptions.

Polyspace considers exceptions of the class `std::exception` as generic.These are base exception classes that have many derived exception classes.

### Risk

Raising generic exceptions hinders proper error handling and error recovery. For instance:

```
void foo(void) throw(std::exception){
    //...
}

void caller(void){
    try{
        //...
        foo();
    }catch(std::exception& e){
        // Need to figure out what the error is.
    }
}
```

When handling the exceptions raised by `foo()`, determining what errors might happen in `foo()` is handled by `caller()`. Because the exception specification of `foo()` is broad, the emergence of new or unexpected exceptions might not be detected. Handling and recovering from errors in `foo()` becomes more complicated for `caller()`.

### Fix

To resolve this defect, use specific exceptions when documenting the exception in a function. Caller functions are then able to address specific exceptions by using specific error-handling code. Unexpected exceptions are easily detected.

## Examples

### Avoid Generic Exception Specification

```
#include<exception>
#include<stdexcept>
```

```
#include<iostream>

void foo(void) throw(std::exception){
    //...
}
// @throw std::exception
void bar() {  //Noncompliant
    //...
    throw std::exception();
}

void foo1(){
    //...
    throw std::exception();
}
int main(void){
    try{
        //...
        foo();
    }catch(std::exception& e){
        // Need to figure out what the error is.

    }
    try{
        //...
        bar();
    }catch(std::exception& e){
        // Need to figure out what the error is.

    }
    try{
        //...
        foo1();
    }catch(std::exception& e){
        // Need to figure out what the error is.

    }
}
```

In this example, Polyspace flags the generic `throw` specifications. For instance:

- Polyspace flags the dynamic exception specification of `foo()` because it uses `std::exception`.
- Polyspace flags the function `bar()` because it is documented to raise the generic exception `std::exception`. Polyspace also flags the generic `throw` statement in this function.
- Polyspace flags the generic `throw` statement in `foo1()`.

Because these functions raise generic exceptions, identifying the exceptions is handled by the caller function `main()`. Because the caller does not know which specific exception are expected, it might fail to detect unexpected exceptions.

#### Correction — Use Specific Exceptions

To fix this issue, document specific exceptions that might arise from a function. This specific documentation enables the caller function to address specific exceptions. Unexpected exceptions become unhandled exceptions and are easily detected.

```
#include<exception>
#include<stdexcept>
```

```
#include<iostream>

void foo(void) throw(std::overflow_error){
    //...
}
// @throw std::invalid_argument
void bar() {
    //...
}

void foo1(){
    //...
    throw std::range_error("MSG");
}
int main(void){
    try{
        //...
        foo();
    }catch(std::overflow_error& e){
        // Expecting an overflow.

    }
    try{
        //...
        bar();
    }catch(std::invalid_argument& e){
        // Expecting an invalid argument.

    }
    try{
        //...
        foo1();
    }catch(std::range_error& e){
        // Expecting a range error.

    }
}
```

## Result Information

**Group:** Good practice
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `THROW_FOR_GENERIC_EXCEPTION`
**Impact:** Low

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”

"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Public static field not `const`

A `static` and `public` field of a `struct` or `class` is not marked as a `const`

## Description

This defect occurs when all of these conditions are true:

- A class contains a data member that is both `static` and `public`
- The `static` and `public` data member is not `const`

### Risk

The `public` `static` data members of a class can be modified by any of class in the program. Unless `public` `static` data members are specified as `const`, they might be modified in unexpected ways and result in bugs that are difficult to diagnose.

### Fix

To resolve this defect, specify the `public static` data members as `const` or `constexpr`.

## Examples

### Avoid Nonconst `public static` Data Members

```
#include<string>
class SomeAppClass {

private:
    static std::string appPropertiesConfigFilep2;

protected:
    static std::string appPropertiesConfigFilepr2;

public:
    static std::string appPropertiesConfigFile2; //Defect

    static void foo() {}
    int bar() {
        static int v = 3;
        return v;
    }
};
```

In this example, the data member `appPropertiesConfigFile2` is `static` and `public`, but not `const`. Polyspace flags the member. The `static` data members that are `private` and `protected` are not flagged. This rule does not apply to `static` objects that are in the scope of `public` member functions of a class.

#### Correction — Declare `public static` Data Members as `const`

To resolve this defect, declare the `public static` data members of a class as `const`.

```
#include<string>
class SomeAppClass {

private:
    static std::string appPropertiesConfigFilep2;

protected:
    static std::string appPropertiesConfigFilepr2;

public:
    static const std::string appPropertiesConfigFile2; //No Defect

    static void foo() {}
    int bar() {
        static int v = 3;
        return v;
    }
};
```

## Result Information

**Group:** Good practice
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `PUBLIC_STATIC_FIELD_NOT_CONST`
**Impact:** Low

# Version History

**Introduced in R2022a**

## See Also

`Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Method not `const`

A method that can be made `const` is not marked `const`

## Description

This defect occurs when a method that meets these criteria is not marked as a `const`:

- The method is not `virtual`.
- The method does not return pointers or references to nonconst data.
- The method does not modify any variables, does not call global functions, and does not call any nonconst member functions.

### Risk

If a method in your code satisfies the requirements of being a `const` method, then declaring the method as a `const` has these advantages:

- Objects that are `const` are not allowed to invoke nonconst methods. When you declare a method as `const`, some additional references and objects in your code might be declared as `const`, increasing the `const` correctness of your code.
- The C++11 standard assumes that `const` methods are free of data races. Using such methods in multiple threads does not require external synchronization, for instance, by using `mutex` objects. Using `const` method makes concurrent programming convenient.

Not declaring eligible methods a `const` makes maintaining the `const` correctness of your code more difficult. Code that is not const-correct remains vulnerable to unexpected mutation of variables, unexpected assignments, and inefficient compiler optimization.

### Fix

To fix this defect, declare eligible methods as `const`. If a `const` overload of the function already exists, consider removing the nonconst overload.

## Examples

### Declare Methods as `const` if Applicable

```
 class LossyPolymer{
    double n;
    double k;
    public:
    LossyPolymer(double n_in, double k_in):n(n_in), k(k_in){}
    double getN() {
        return n;
    }
    double getk() {
        return k;
    }

};
```

```
void foo(){
    LossyPolymer PMMA = LossyPolymer(1.5,0.001);
    const LossyPolymer PVK = LossyPolymer(1.6,0.01);
    int k_PMMA = PMMA.getk();
    //int k_PVK = PVK.getk(); Compile error
}
```

In this example, a class `LossyPolymer` is implemented that represents the optical properties of polymer materials. The optical properties of a `LossyPolymer` is represented by the private variables `n` and `k`. The class has getter methods for these variables that return a copy of the private variables. The getter methods of `LossyPolymer` do not modify any variable, but they are not declared `const`, perhaps inadvertently.

The class `LossyPolymer` represents a physical material. A physical material has constant optical properties. Objects of the class `LossyPolymer` are expected to be `const` objects, such as `PVK`. Because `getk()` is nonconst, the `const` object `PVK` cannot call its `getK()` method. To use the getter, you must declare instances of `LossyPolymer` as nonconst objects, which is not the best practice. The incorrect constness of the getter function prevents const-correct behavior in the code. Polyspace flags them.

#### Correction — Declare Eligible Methods as `const`

To fix this defect, declare the getter methods as `const`. Doing so allows you to declare `PVK` and `PMMA` as `const` objects and call the getter functions.

```
 class LossyPolymer{
    double n;
    double k;
    public:
    LossyPolymer(double n_in, double k_in):n(n_in), k(k_in){}
    const double getN() const{
        return n;
    }
    const double getk() const{
        return k;
    }

};

void foo(){
    LossyPolymer PMMA = LossyPolymer(1.5,0.001);
    const LossyPolymer PVK = LossyPolymer(1.6,0.01);
    int k_PMMA = PMMA.getk();
    int k_PVK = PVK.getk();
}
```

## Result Information

**Group:** Good practice
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `METHOD_NOT_CONST`
**Impact:** Low

## Version History

**Introduced in R2022a**

## See Also

`Variable shadowing` | `Non-initialized variable` | `Write without a further read` | `Improper array initialization` | `Find defects (-checkers)`

### Topics

"Interpret Bug Finder Results in Polyspace Desktop User Interface"
"Interpret Bug Finder Results in Polyspace Access Web Interface"
"Address Results in Polyspace User Interface Through Bug Fixes or Justifications"
"Address Results in Polyspace Access Through Bug Fixes or Justifications"

# Reference to un-named temporary

Local reference is declared by using unnamed temporary variable that is returned by value

## Description

This defect occurs when:

1. A function returns an unnamed temporary object by value.
2. You declare a local reference by using the returned temporary object.

Consider this code:

```
std::string get();
const std::string &s = get();
```

The function `get()` returns a string by value. The local reference `s` is declared by using the unnamed temporary string. Polyspace reports a defect.

### Risk

Returning an object by copy and then declaring a local reference by using the returned copy can lead to unexpected behavior. For instance, If you later update the code to return the temporary object by reference, perhaps to improve performance, the local reference then resolves to a nonlocal object instead of a local copy. Modifying the nonlocal object causes unexpected results. See “Local Reference Declared by Using Unnamed Returned-by-Value Temporary Object” on page 20-85.

This defect can lead to inefficiencies in cases where it is possible to use the unnamed temporary object directly instead of using a reference to the temporary objeect. See “Local Reference Used for Passing Object to Function” on page 20-87.

### Fix

To fix this defect, capture the returned unnamed temporary object by using a variable instead of a reference. When possible, use the unnamed temporary object directly.

## Examples

### Local Reference Declared by Using Unnamed Returned-by-Value Temporary Object

In this example, the function `f.getname()` returns an unnamed temporary `std::string` object by value. The function `func` declares the local reference `s` by using this temporary string object. Polyspace reports a defect.

```
#include<string>
class myString
{
    public:
        myString(const std::string &s): m_name(s) {}

        std::string getName() const { return m_name; }
        void setName(const std::string &s) { m_name = s; }
    private:
        std::string m_name;
};

void func()
{
    myString f("Brian");

    const std::string &s = f.getName();

    f.setName("Bob");
    assert(s == "Brian"); //Succeeds
}
```

Suppose that to improve efficiency, you update the function `myString::getname` to return an `std::string` object by reference. Then, the reference `s` resolves to the string `myString::m_name` instead of a copy of the string. If your code relies on the fact that `s` is a copy of `myString::m_name`, then the code might produce unexpected results after the change to `myString::getname`. For instance, in the preceding code, `s` contains `Brian`, but in the following code, `s` contains `Bob`, which causes the `assert()` statement to fail.

```
#include<string>
class myString
{
    public:
        myString(const std::string &s): m_name(s) {}

        //Updated getname() for improved performance

        const std::string &getName() const { return m_name; }
        void setName(const std::string &s) { m_name = s; }

    private:
        std::string m_name;
};
void func()
{
    myString f("Brian");

    const std::string &s = f.getName();

    f.setName("Bob");
    assert(s == "Brian");//Fails!
}
```

**Correction — Avoid Using Local References to Store Returned-by-Value Objects**

To fix this defect, capture the returned-by-value temporary string in a variable instead of a reference.

```
#include<string>
class myString
{
    public:
        myString(const std::string &s): m_name(s) {}

        std::string getName() const { return m_name; }
        void setName(const std::string &s) { m_name = s; }
    private:
        std::string m_name;
};

void func()
{
    myString f("Brian");

    const std::string s = f.getName(); //No defect

    f.setName("Bob");
    assert(s == "Brian"); //Succeeds
}
```

The `assert` statement succeeds if `myString::getname` is later changed to return an `std::string` by reference.

#### Local Reference Used for Passing Object to Function

In some cases, it is more efficient to use the unnamed temporary object directly instead of capturing the object in a reference. For instance, in this code, Polyspace reports a defect.

```
#include<string>
extern       std::string  getSByValue();
extern void useString(std::string s);

void func()
{
    const std::string &s = getSByValue();
    useString(s);
}
```

##### Correction — Use Unnamed Temporary Directly

To fix this issue, use the unnamed temporary object directly when calling `useString()`. In this case, capturing the temporary object is unnecessary.

```
#include<string>
extern       std::string  getSByValue();
extern void useString(std::string s);

void func()
{
    useString(getSByValue());
}
```

## Result Information

**Group:** Good practice
**Language:** C++
**Default:** Off
**Command-Line Syntax:** `LOCAL_REF_TO_UNNAMED_TEMPORARY`
**Impact:** Low

## Version History

**Introduced in R2023a**

## See Also

`Find defects (-checkers)`

### Topics

“Interpret Bug Finder Results in Polyspace Desktop User Interface”
“Interpret Bug Finder Results in Polyspace Access Web Interface”
“Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
“Address Results in Polyspace Access Through Bug Fixes or Justifications”

# Polyspace Results: Coding Standards

21

# MISRA C 2012

# MISRA C:2012 Dir 1.1

Any implementation-defined behavior on which the output of the program depends shall be documented and understood

## Description

### Directive Definition

*Any implementation-defined behavior on which the output of the program depends shall be documented and understood.*

### Rationale

A code construct has implementation-defined behavior if the C standard allows compilers to choose their own specifications for the construct. The full list of implementation-defined behavior is available in Annex J.3 of the standard ISO/IEC 9899:1999 (C99) and in Annex G.3 of the standard ISO/IEC 9899:1990 (C90).

If you understand and document all implementation-defined behavior, you can be assured that all output of your program is intentional and not produced by chance.

### Polyspace Implementation

The analysis detects the following possibilities of implementation-defined behavior in C99 and their counterparts in C90. If you know the behavior of your compiler implementation, justify the analysis result with appropriate comments. To justify a result, assign one of these statuses: `Justified`, `No action planned`, or `Not a defect`.

**Tip** To mass-justify all results that indicate the same implementation-defined behavior, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

| C99 Standard Annex Ref | Behavior to Be Documented | How Polyspace Helps |
|---|---|---|
| J.3.2: Environment | An alternative manner in which `main` function may be defined. | The analysis flags `main` with arguments and return types other than:<br>`int main(void) { ... }`<br>or<br>`int main(int argc, char *argv[]) { ... }`<br>See section 5.1.2.2.1 of the C99 Standard. |

| C99 Standard Annex Ref | Behavior to Be Documented | How Polyspace Helps |
|---|---|---|
| J.3.2: Environment | The set of environment names and the method for altering the environment list used by the `getenv` function. | The analysis flags uses of the `getenv` function. For this function, you need to know the list of environment variables and how the list is modified.<br>See section 7.20.4.5 of the C99 Standard. |
| J.3.6: Floating Point | The rounding behaviors characterized by non-standard values of `FLT_ROUNDS`. | The analysis flags the include of `float.h` if values of `FLT_ROUNDS` are outside the set, {-1, 0, 1, 2, 3}. Only the values in this set lead to well-defined rounding behavior.<br>See section 5.2.4.2.2 of the C99 Standard. |
| J.3.6: Floating Point | The evaluation methods characterized by non-standard negative values of `FLT_EVAL_METHOD`. | The analysis flags the include of `float.h` if values of `FLT_EVAL_METHOD` are outside the set, {-1, 0, 1, 2}. Only the values in this set lead to well-defined behavior for floating-point operations.<br>See section 5.2.4.2.2 of the C99 Standard. |
| J.3.6: Floating Point | The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value. | The analysis flags conversions from integer to floating-point data types of smaller size (for example, 64-bit `int` to 32-bit `float`).<br>See section 6.3.1.4 of the C99 Standard. |
| J.3.6: Floating Point | The direction of rounding when a floating-point number is converted to a narrower floating-point number. | The analysis flags these conversions:<br>• `double` to `float`<br>• `long double` to `double` or `float`<br>See section 6.3.1.5 of the C99 Standard. |
| J.3.6: Floating Point | The default state for the `FENV_ACCESS` pragma. | The analysis flags use of the pragma other than:<br>`#pragma STDC FENV_ACCESS ON`<br>or<br>`#pragma STDC FENV_ACCESS OFF`<br>See section 7.6.1 of the C99 Standard. |

| C99 Standard Annex Ref | Behavior to Be Documented | How Polyspace Helps |
|---|---|---|
| J.3.6: Floating Point | The default state for the `FP_CONTRACT` pragma. | The analysis flags use of the pragma other than:<br>`#pragma STDC FP_CONTRACT ON`<br>or<br>`#pragma STDC FP_CONTRACT OFF`<br>See section 7.12.2 of the C99 Standard. |
| J.3.11: Preprocessing Directives | The behavior on each recognized non-STDC #pragma directive. | The analysis flags the pragma usage:<br>`#pragma` *`pp-tokens`*<br>where the processing token `STDC` does not immediately follow`pragma`. For instance:<br>`#pragma FENV_ACCESS ON`<br>See section 6.10.6 of the C99 Standard. |
| J.3.12: Library Functions | Whether the `feraiseexcept` function raises the ‘‘inexact’’ floating-point exception in addition to the ‘‘overflow’’ or ‘‘underflow’’ floating-point exception. | The analysis flags calls to the `feraiseexcept` function.<br>See section 7.6.2.3 of the C99 Standard. |
| J.3.12: Library Functions | Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function. | The analysis flags calls to the `setlocale` function when its second argument is not `"C"` or `""`.<br>See section 7.11.1.1 of the C99 Standard. |
| J.3.12: Library Functions | The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2. | The analysis flags the include of `math.h` if `FLT_EVAL_METHOD` has values outside the set {0,1,2}.<br>See section 7.12 of the C99 Standard. |

| C99 Standard Annex Ref | Behavior to Be Documented | How Polyspace Helps |
|---|---|---|
| J.3.12: Library Functions | The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient. | The analysis flags calls to the `remquo`, `remquof` and `remquol` function.<br>See section 7.12.10.3 of the C99 Standard. |
| J.3.12: Library Functions | The termination status returned to the host environment by the `abort`, `exit`, or `_Exit` function. | The analysis flags calls to the `abort`, `exit`, or `_Exit` function.<br>See sections 7.20.4.1, 7.20.4.3 or 7.20.4.4 of the C99 Standard. |

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** The implementation
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2017b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Dir 2.1

All source files shall compile without any compilation errors

## Description

### Directive Definition

*All source files shall compile without any compilation errors*.

### Rationale

A conforming compiler is permitted to produce an object module despite the presence of compilation errors. However, execution of the resulting program can produce unexpected behavior.

### Polyspace Implementation

The software raises a violation of this directive if it finds a compilation error.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Compilation and build
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2015b**

## See Also

`MISRA C:2012 Rule 1.1` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Dir 4.1

Run-time failures shall be minimized

## Description

### Directive Definition

*Run-time failures shall be minimized.*

### Rationale

To optimize the size and speed of executable code, the C standard implements limited run-time failure checks. To reduce run-time failures, check these errors:

- Arithmetic errors
- Pointer arithmetic
- Array bound errors
- Function parameters
- Pointer dereferencing
- Dynamic memory

### Polyspace Implementation

Polyspace reports a violation of this directive if runtime issues exist in your code, including:

- `Integer division by zero`
- `Float division by zero`
- `Integer conversion overflow`
- `Unsigned integer conversion overflow`
- `Sign change integer conversion overflow`
- `Float conversion overflow`
- `Integer overflow`
- `Unsigned integer overflow`
- `Float overflow`
- `Shift of a negative value`
- `Right operand of shift operation outside allowed bounds`
- `Dereference of a null pointer`
- `Arithmetic operation with NULL pointer`
- `Automatic or thread local variable escaping from a thread`
- `Wrong allocated object size for cast`
- `Array access out of bounds`
- `Pointer access out of bounds`

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Code design
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Dir 4.11` | `MISRA C:2012 Rule 1.3` | `MISRA C:2012 Rule 18.1` | `MISRA C:2012 Rule 18.2` | `MISRA C:2012 Rule 18.3` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Dir 4.3

Assembly language shall be encapsulated and isolated

## Description

### Directive Definition

*Assembly language shall be encapsulated and isolated*.

### Rationale

Encapsulating assembly language is beneficial because:

- Encapsulating assembly instructions in a C source files clearly marks the boundary between C and assembly, making the code easier to read.
- The name, and documentation, of the encapsulating macro or function makes the intent of the assembly language clear.
- All uses of assembly language for a given purpose can share encapsulation, which improves maintainability.
- You can easily substitute the assembly language for a different target or for static analysis purposes.

### Polyspace Implementation

Polyspace reports a violation of this rule if assembly language code is used without encapsulation. It is not a violation if the assembly code is encapsulated in:

- Assembly language functions. Consider this code:

  ```
  asm int h(int tt)  //Compliant
  {
    % reg val;
    mtmsr val;
    return 3;
  };

  void f(void) {
    int x;
    x = h(3);
  }
  ```

  The function `h()` is declared as an `asm` function which encapsulate the assembly code. Polyspace does not report a violation.
- `#pragma` directives. For example, in this code:

  ```
  #pragma inline_asm(h)
  int h(int tt)  //Compliant
  {
    % reg val;
    mtmsr val;
    return 3;
  };
  ```

```
void f(void) {
  int x;
  x = h(3);
}
```

The `#inline_asm` pragma designates the function `h()` as an assembly language function. Polyspace does not report a violation.

- Macros. For instance:

```
#define FUNC_H\
asm\
{\
 % reg val; \
  mtmsr val;\
  return 3; \
}; \

void f(void) {
  int x;
  x = FUNC_H(3);
}
```

The macro `FUNC_H` encapsulates the assembly code. Polyspace does not report a violation. You can also define such macros at the command line during the Polyspace analysis by using the options `Preprocessor definitions (-D)`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Assembly Language Code in C Function

```
enum boolVal {TRUE, FALSE};
enum boolVal isTaskActive;
void taskHandler(void);

void taskHandler(void) {
    isTaskActive = FALSE;
    // Software interrupt for task switching
    asm volatile   /* Non-compliant */
    (
        "SWI &02"     /* Service #1: calculate run-time */
    );
    return;
}
```

In this example, the rule violation occurs because the assembly language code is embedded directly in a C function `taskHandler` that contains other C language statements.

#### Correction — Encapsulate Assembly Code in Macro

One correction is to encapsulate the assembly language code in a macro and invoke the macro in the function `taskHandler`.

```
#define  RUN_TIME_CALC \
asm volatile \
    ( \
        "SWI &02"     /* Service #1: calculate run-Time */ \
    )\

enum boolVal {TRUE, FALSE};
enum boolVal isTaskActive;
void taskHandler(void);

void taskHandler(void) {
    isTaskActive = FALSE;
    RUN_TIME_CALC;
    return;
}
```

### Check Information

**Group:** Code design
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

### See Also

`MISRA C:2012 Rule 1.2` | `Check MISRA C:2012 (-misra3)`

**Topics**

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Dir 4.4

Sections of code should not be "commented out"

## Description

### Directive Definition

*Sections of code should not be "commented out"*.

### Rationale

C comments enclosed in `/* */` do not support nesting. A comment beginning with `/*` ends at the first `*/` even when the `*/` is intended as the end of a later nested comment. If a section of code that is commented out already contains comments, you can encounter compilation errors (or at least comment out less code than you intend).

Commenting out code is not a good practice. The commented out code can remain out of sync with the surrounding code without causing compilation errors. Later, if you uncomment the code, you can encounter unexpected issues.

Use comments only to explain aspects of the code that are not apparent from the code itself.

### Polyspace Implementation

The checker uses internal heuristics to detect commented out code. For instance, characters such as `#`, `;`, `{` or `}` indicate comments that might potentially contain code. These comments are then evaluated against other metrics to determine the likelihood of code masquerading as comment. For instance, several successive words without a symbol in between reduces this likelihood.

The checker does not flag the following comments even if they contain code:

- Doxygen comments beginning with `/**`, `/*!`, `///` or `//!`.
- Comments that repeat the same symbol more than five times, for instance, the symbol = here:

  ```
  /* =====================================
   * A comment
   * =====================================*/
  ```
- Comments on the first line of a file.
- Comments that mix the C style (`/* */`) and C++ style (`//`).
- Comments that contain one or more `@` symbol. If the `@` symbol is placed in a nested comment that contains code, Polyspace flags it. For instance:.

  ```
  int* q;
  //@Error int foo(void);
  //...
  void bar(void){
      /*
      int*p =  (int*) malloc(int); // Error @allocation
      */
  }
  ```

In the preceding code, Polyspace flags the second comment block containing the commented out `malloc` operation, and ignores the first comment.

The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Code Commented Out

```
#include <stdlib.h>
/* =====================================
 * usage:print32_tInteger();
 * =====================================*/
int32_t getRandInt();
void print32_t(int32_t);

//Error@ int32_t val = getRandInt();
void print32_tInteger() {
    /* int32_t val = getRandInt(); //Noncompliant
     * val++;  // contact support @..
     * print32_t(val); */
    print32_t(getRandInt());
}
```

This example contains several comments that contains code.

- The first comment block documents the usage of the function `print32_tInteger()`. Because the comment uses the symbol = more than five times, Polyspace does not flag this comment.
- The second comment documents the source of error in the code. Because the code contains the symbol `@`, Polyspace ignores the comment.
- The third comment block comments out code that might contain errors. This comment does not document anything and simply excludes code from compilation. Polyspace flags this code block. Because the `@` symbol is in a nested comment, Polyspace does not ignore the comment.

## Check Information

**Group:** Code design
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2020b**

## See Also

`Check MISRA C:2012 (-misra3)`

**Topics**

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Dir 4.5

Identifiers in the same name space with overlapping visibility should be typographically unambiguous

## Description

### Directive Definition

*Identifiers in the same name space with overlapping visibility should be typographically unambiguous.*

### Rationale

What “unambiguous” means depends on the alphabet and language in which source code is written. When you use identifiers that are typographically close, you can confuse between them.

For the Latin alphabet as used in English words, at a minimum, the identifiers should not differ by:

- The interchange of a lowercase letter with its uppercase equivalent.
- The presence or absence of the underscore character.
- The interchange of the letter `O` and the digit `0`.
- The interchange of the letter `I` and the digit `1`.
- The interchange of the letter `I` and the letter `l`.
- The interchange of the letter `S` and the digit `5`.
- The interchange of the letter `Z` and the digit `2`.
- The interchange of the letter `n` and the letter `h`.
- The interchange of the letter `B` and the digit `8`.
- The interchange of the letters `rn` and the letter `m`.

### Polyspace Implementation

The checker flags identifiers in the same scope that differ from each other only in the above characters. Polyspace ignores identifiers that are defined in macros starting with `do`, `for`, `switch`, and `while`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val;  /* Non-compliant */

    int id2_numval;
    int id2_numVal;   /* Non-compliant */

    int id3_lvalue;
    int id3_Ivalue;   /* Non-compliant */

    int id4_xyZ;
    int id4_xy2;      /* Non-compliant */

    int id5_zerO;
    int id5_zer0;     /* Non-compliant */

    int id6_rn;
    int id6_m;        /* Non-compliant */
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

## Check Information

**Group:** Code design
**Category:** Advisory
**AGC Category:** Readability

# Version History

**Introduced in R2015b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Dir 4.6

`typedefs` that indicate size and signedness should be used in place of the basic numerical types

## Description

### Directive Definition

*`typedefs` that indicate size and signedness should be used in place of the basic numerical types.*

### Rationale

When the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object.

### Polyspace Implementation

The rule checker flags use of basic data types in variable or function declarations and definitions. The rule enforces use of `typedefs` instead.

The rule checker does not flag the use of basic types in the `typedef` statements themselves.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Direct Use of Basic Types in Definitions

```
typedef unsigned int uint32_t;

int x = 0;        /* Non compliant */
uint32_t y = 0;  /* Compliant */
```

In this example, the declaration of `x` is noncompliant because it uses a basic type directly.

## Check Information

**Group:** Code design
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

**Topics**

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Dir 4.7

If a function returns error information, then that error information shall be tested

## Description

### Directive Definition

*If a function returns error information, then that error information shall be tested*.

### Rationale

If you do not check the return value of functions that indicate error information through their return values, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

### Polyspace Implementation

The checker raises a violation when you call sensitive standard functions that return information about possible errors and you do one of the following:

- Ignore the return value.

  You simply do not assign the return value to a variable, or explicitly cast the return value to `void`.
- Use an output from the function (return value or argument passed by reference) without testing the return value for errors.

The checker considers a function as sensitive if the function call is prone to failure because of reasons such as:

- Exhausted system resources (for example, when allocating resources).
- Changed privileges or permissions.
- Tainted sources when reading, writing, or converting data from external sources.
- Unsupported features despite an existing API.

The checker only considers functions where the *return value* indicates if the function completed without errors.

Some of these functions can perform critical tasks such as:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

For functions that are not critical, the checker allows casting the function return value to `void`.

This directive is only partially supported.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Sensitive Function Return Ignored

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>
#include <cstdlib>
#define fatal_error() abort()

void initialize_1() {
    pthread_attr_t attr;
    pthread_attr_init(&attr); //Noncompliant
}

void initialize_2() {
    pthread_attr_t attr;
   (void)pthread_attr_init(&attr); //Compliant
}

void initialize_3() {
    pthread_attr_t attr;
    int result;
    result = pthread_attr_init(&attr); //Compliant
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}

int read_file_1(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r");
  return 0; //Noncompliant

}
int read_file_2(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r"); //Compliant
  if (in==NULL){
      // Handle error
  }
  return 0;
}
```

This example shows a call to the sensitive functions `pthread_attr_init` and `fmemopen`. Polyspace raises a flag if:

- You implicitly ignore the return of the sensitive function. Explicitly ignoring the output of sensitive functions is not flagged.
- You obtain the return value of a sensitive function but do not test the value before exiting the relevant scope. The violation is raised on the exit statement.

To be compliant, you can explicitly cast their return value to `void` or test the return values to check for errors.

### Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked_1() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0)); //Noncompliant
    pthread_join(thread_id,  &res); //Noncompliant
}

void returnnotchecked_2() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL); //Compliant
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res); //Compliant
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), the rule checker still raises a violation. The other critical function, `pthread_join`, returns a value that is ignored implicitly.

To be compliant, check the return value of these critical functions to verify the function performed as expected.

## Check Information

**Group:** Code design
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2017a**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Dir 4.8

If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden

## Description

### Rule Definition

*If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.*

### Rationale

If a pointer to a structure or union is not dereferenced in a file, the implementation details of the structure or union need not be available in the translation unit for the file. You can hide the implementation details such as structure members and protect them from unintentional changes.

Define an opaque type that can be referenced via pointers but whose contents cannot be accessed.

### Polyspace Implementation

If a structure or union is defined in a file or a header file included in the file, a pointer to this structure or union declared but the pointer never dereferenced in the file, the checker flags a coding rule violation. The structure or union definition should not be visible to this file.

If you see a violation of this rule on a structure definition, identify if you have defined a pointer to the structure in the same file or in a header file included in the file. Then check if you dereference the pointer anywhere in the file. If you do not dereference the pointer, the structure definition should be hidden from this file and included header files.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Object Implementation Revealed

`file.h`: Contains structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct  {
  int a;
} myStruct;

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

myStruct* getObj(void);
void useObj(myStruct*);

void func() {
  myStruct *sPtr = getObj();
  useObj(sPtr);
}
```

In this example, the pointer to the type `myStruct` is not dereferenced. The pointer is simply obtained from the `getObj` function and passed to the `useObj` function.

The implementation of `myStruct` is visible in the translation unit consisting of `file.c` and `file.h`.

**Correction — Define Opaque Type**

One possible correction is to define an opaque data type in the header file `file.h`. The opaque data type `ptrMyStruct` points to the `myStruct` structure without revealing what the structure contains. The structure `myStruct` itself can be defined in a separate translation unit, in this case, consisting of the file `file2.c`. The common header file `file.h` must be included in both `file.c` and `file2.c` for linking the structure definition to the opaque type definition.

`file.h`: Does not contain structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct myStruct *ptrMyStruct;

ptrMyStruct getObj(void);
void useObj(ptrMyStruct);

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

void func() {
  ptrMyStruct sPtr = getObj();
  useObj(sPtr);
}
```

`file2.c`: Includes `file.h` and dereferences structure.

```
#include "file.h"

struct myStruct {
  int a;
};

void useObj(ptrMyStruct ptr) {
    (ptr->a)++;
}
```

## Check Information

**Group:** Code design<br>
**Category:** Advisory<br>
**AGC Category:** Advisory

## Version History

**Introduced in R2018a**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Dir 4.9

A function should be used in preference to a function-like macro where they are interchangeable

## Description

### Directive Definition

*A function should be used in preference to a function-like macro where they are interchangeable.*

### Rationale

In most circumstances, use functions instead of macros. Functions perform argument type-checking and evaluate their arguments once, avoiding problems with potential multiple side effects.

### Polyspace Implementation

Polyspace considers all function-like macro definitions.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Code design
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 13.2` | `MISRA C:2012 Rule 20.7` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Dir 4.12

Dynamic memory allocation shall not be used

## Description

### Rule Definition

*Dynamic memory allocation shall not be used.*

### Rationale

Using dynamic memory allocation and deallocation routines provided by the Standard Library or third-party libraries can cause undefined behavior. For instance:

- You use `free` to deallocate memory that you did not allocate with `malloc`, `calloc`, or `realloc`.
- You use a pointer that points to a freed memory location.
- You access allocated memory that has no value stored into it.

Dynamic memory allocation and deallocation routines from third-party libraries are likely to exhibit similar undefined behavior.

If you choose to use dynamic memory allocation and deallocation routines, ensure that your program behavior is predictable. For example, ensure that you safely handle allocation failure due to insufficient memory.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `malloc`, `calloc`, `realloc` and `free`

```
#include <stdlib.h>

static int foo(void);

typedef struct struct_1 {
    int a;
    char c;
} S_1;

static int foo(void) {

    S_1 * ad_1;
    int  * ad_2;
    int  * ad_3;

    ad_1 = (S_1*)calloc(100U, sizeof(S_1));        /* Non-compliant */
    ad_2 = malloc(100U * sizeof(int));             /* Non-compliant */
    ad_3 = realloc(ad_3, 60U * sizeof(long));      /* Non-compliant */
    free(ad_1);                                    /* Non-compliant */
    free(ad_2);                                    /* Non-compliant */
    free(ad_3);                                    /* Non-compliant */

    return 1;
}
```

In this example, the rule is violated when the functions `malloc`, `calloc`, `realloc` and `free` are used.

## Check Information

**Group:** Code Design
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2019b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Dir 4.10

Precautions shall be taken in order to prevent the contents of a header file being included more than once

## Description

### Directive Definition

*Precautions shall be taken in order to prevent the contents of a header file being included more than once*.

### Rationale

When a translation unit contains a complex hierarchy of nested header files, it is possible for a particular header file to be included more than once, leading to confusion. If this multiple inclusion produces multiple or conflicting definitions, then your program can have undefined or erroneous behavior.

For instance, suppose that a header file contains:

```
#ifdef _WIN64
   int env_var;
#elseif
   long int env_var;
#endif
```

If the header file is contained in two inclusion paths, one that defines the macro `_WIN64` and another that undefines it, you can have conflicting definitions of `env_var`.

### Polyspace Implementation

If you include a header file whose contents are not guarded from multiple inclusion, the analysis raises a violation of this directive. The violation is shown at the beginning of the header file.

You can guard the contents of a header file from multiple inclusion by using one of the following methods:

```
<start-of-file>
#ifndef <control macro>
#define <control macro>
    /* Contents of file */
#endif
<end-of-file>
```

or

```
<start-of-file>
#ifdef <control macro>
#error ...
#else
#define <control macro>
```

```
    /* Contents of file */
#endif
<end-of-file>
```

Unless you use one of these methods, Polyspace flags the header file inclusion as noncompliant.

#### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Code After Macro Guard

```
#ifndef  __MY_MACRO__
#define __MY_MACRO__
    void func(void);
#endif
void func2(void);
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func2(void)` is outside the guard.

**Note** You can have comments outside the macro guard.

### Code Before Macro Guard

```
void func(void);
#ifndef  __MY_MACRO__
#define __MY_MACRO__
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func(void)` is outside the guard.

**Note** You can have comments outside the macro guard.

### Mismatch in Macro Guard

```
#ifndef  __MY_MACRO__
#define __MY_MARCO__
    void func(void);
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro name in the `#ifndef` statement is different from the name in the following `#define` statement.

## Check Information

**Group:** Code Design
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Dir 4.11

The validity of values passed to library functions shall be checked

## Description

### Directive Definition

*The validity of values passed to library functions shall be checked*.

### Rationale

Many Standard C functions do not check the validity of parameters passed to them. Even if checks are performed by a compiler, there is no guarantee that the checks are adequate. For example, you should not pass negative numbers to `sqrt` or `log`.

### Polyspace Implementation

Polyspace reports violation of this directive if any of these issues are detected:

- `Invalid use of standard library floating point routine`.
- `Invalid use of standard library routine`.
- `Standard function call with incorrect arguments`.
- `Invalid use of standard library integer routine`. Polyspace does not report a violation of this directive on invalid use of some integer library routines such as `isalnum`, `isalpha`, or `tolower`.
- `Invalid use of standard library memory routine`. Polyspace does not report a violation of this directive on invalid use of some memory library routines such as `memchr`, `memcmp`, or `memset`.
- `Invalid use of standard library string routine`. Polyspace does not report a violation of this directive on invalid use of some string library routines such as `strncat`, `strncpy`, or `strcpy`.

**Tip** To mass-justify all results related to the same library function, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Code design<br>
**Category:** Required

**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Dir 4.1` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Dir 4.13

Functions which are designed to provide operations on a resource should be called in an appropriate sequence

## Description

### Directive Definition

*Functions which are designed to provide operations on a resource should be called in an appropriate sequence*.

### Rationale

You typically use functions operating on a resource in the following way:

1. You allocate the resource.

   For example, you open a file or critical section.
2. You use the resource.

   For example, you read from the file or perform operations in the critical section.
3. You deallocate the resource.

   For example, you close the file or critical section.

For your functions to operate as you expect, perform the steps in sequence. For instance, if you call a resource allocation function on a certain execution path, you must call a deallocation function on that path.

### Polyspace Implementation

Polyspace Bug Finder detects a violation of this rule if you specify multitasking options and your code contains one of these defects:

- `Missing lock`: A task calls an unlock function before calling the corresponding lock function.
- `Missing unlock`: A task calls a lock function but ends without a call to the corresponding unlock function.
- `Double lock`: A task calls a lock function twice without an intermediate call to an unlock function.
- `Double unlock`: A task calls an unlock function twice without an intermediate call to a lock function.

For more information on the multitasking options, see “Multitasking”.

### Extend Checker

You might be using multithreading functions that are not supported by Polyspace. Extend this checker by mapping the functions of your multithreading functions to their known POSIX equivalent. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Multitasking: Lock Function That Is Missing Unlock Function

```
typedef signed int int32_t;
typedef signed short int16_t;

typedef struct tag_mutex_t {
    int32_t value;
} mutex_t;


extern mutex_t mutex_lock   ( void );
extern void  mutex_unlock (  mutex_t m );

extern int16_t x;
void func(void);

void task1(void) {
     func();
}

void task2(void) {
     func();
}

void func ( void ) {
    mutex_t m = mutex_lock ( );  /* Non-compliant */

    if ( x > 0 )  {
        mutex_unlock ( m );
    }  else  {
        /* Mutex not unlocked on this path */
    }
}
```

In this example, the rule is violated when:

- You specify that the functions `mutex_lock` and `mutex_unlock` are paired.

  `mutex_lock` begins a critical section and `mutex_unlock` ends it.
- The function `mutex_lock` is called. However, if `x <= 0`, the function `mutex_unlock` is not called.

To enable detection of this rule violation, you must specify these analysis options.

| Option | Specification |
|---|---|
| **Configure multitasking manually** | ☑ |

| Option | Specification | |
|---|---|---|
| **Entry points** | `task1`<br>`task2` | |
| **Critical section details** | **Starting routine** | **Ending routine** |
| | `mutex_lock` | `mutex_unlock` |

For more information on the options, see:

- `Tasks (-entry-points)`
- `Critical section details (-critical-section-begin -critical-section-end)`

## Check Information

**Group:** Code design
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2015b**

## See Also

`MISRA C:2012 Rule 22.1` | `MISRA C:2012 Rule 22.2` | `MISRA C:2012 Rule 22.6` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Dir 4.14

The validity of values received from external sources shall be checked

## Description

### Directive Definition

*The validity of values received from external sources shall be checked.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

The values originating from external sources can be invalid because of errors or deliberate modification by attackers. Before using the data, you must check the data for validity.

For instance:

- Before using an external input as an array index, you must check if the input can potentially cause an array bounds error.
- Before using an external variable to control a loop, you must check if the variable can potentially result in an infinite loop.

### Polyspace Implementation

The rule checker looks for these issues:

- `Array access with tainted index`.
- `Command executed from externally controlled path`.
- `Execution of externally controlled command`.
- `Host change using externally controlled elements`.
- `Library loaded from externally controlled path`.
- `Loop bounded with tainted value`.
- `Memory allocation with tainted size`.
- `Pointer dereference with tainted offset`.
- `Tainted division operand`.
- `Tainted modulo operand`.
- `Tainted NULL or non-null-terminated string`.
- `Tainted sign change conversion`.
- `Tainted size of variable length array`.
- `Tainted string format`.
- `Use of externally controlled environment variable`.
- `Use of tainted pointer`.
- Using an externally obtained string without a terminating null character in places where a null-terminated string is expected. Such use might result in undefined behavior. For instance, in this

code, the function `printf()` expects string with a terminating null character. Using the character array `str`, which is not terminated by a null character, results in undefined behavior.

```
char str[10];
scanf("%10c", str);
printf("%s",str);//Null terminated string expected
```

- Using an externally obtained indeterminate string. For instance, a string might be indeterminate if you invoke an `fgets()` family function to set the value of the string but the function call fails:

```
char buffer[10];
fgets(buffer, sizeof(buffer), stdin); //buffer is indeterminate if fgets() fails
printf("%s",buffer); // Possible undefined behvior
```

Because the function `printf()` expects a string with a terminating null character, using `buffer` with this function can result in undefined behavior.

#### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

#### Validity of External Values Not Checked

```
#include <stdio.h>

void f1(char from_user[])
{
        char input [128];
        (void) sscanf (from_user, "%128c", input);
        (void) sprintf ("%s", input);/*Noncompliant*/
}
```

In this example, the `sscanf` statement is noncompliant as there is no check to ensure that the user input is null terminated. The subsequent `sprintf` statement that outputs the string can potentially lead to an array bounds error (buffer overrun).

#### Print Invalid External Input String

In this example, the functions `scanf()` and `fgets()` read two `char` arrays and then print the arrays by calling `printf()`. Because the input strings are obtained externally, they might be indeterminate or lack a terminating null character. Printing such strings by using `printf()` results in undefined behavior. Polyspace reports violations of this rule.

```
#include <stdio.h>
void echo_in() {
       //...
    char buffer[10];
    scanf("%10c", buffer);
    //...
    printf("%s", buffer); //Noncompliant - buffer is not null-terminated
    //...
    fgets(buffer, sizeof(buffer), stdin);
      //...
    printf("%s",buffer); //Noncompliant - buffer might be indeterminate
}
```

## Check Information

**Group:** Code design
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2017a**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 1.1

The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation’s translation limits

## Description

### Rule Definition

*The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation’s translation limits*.

### Polyspace Implementation

The rule checker checks for the issues below. Note that:

- The specifications can depend on the version of the C standard used in the analysis. See `C standard version (-c-version)`.
- You can change some of the limits used by the checker using the option `-code-behavior-specifications`. See `-code-behavior-specifications`.

| Issue | C Standard Dependence | Additional Information |
|---|---|---|
| An integer constant falls outside the range of `long int` (if the constant is signed) or `unsigned long int` (if the constant is unsigned). | Checked for C90 only. | The rule checker uses your specifications for the size of a `long int` variable (typically 32 bits). See also `Target processor type (-target)`. |
| An array of size zero is used. | Checked for C90 only. | |
| The number of macros defined in a translation unit exceeds the limit specified in the standard. | Number of macro definitions allowed:<br>• C90: 1024<br>• C99 and later: 4095 | A translation unit consists of a source file together with headers included directly or indirectly in the source file. These are the files necessary to create the smallest object file during compilation. The rule checker requires that the number of macros in a source plus included headers must not exceed the limit specified in the standard. |
| The depth of nesting in control flow statements (like `if`, `while`, etc.) exceeds the limit specified in the standard. | Maximum nesting depth allowed:<br>• C90: 15<br>• C99 and later: 127 | |

| Issue | C Standard Dependence | Additional Information |
|---|---|---|
| The number of levels of inclusion using include files exceeds the limit specified in the standard. | Maximum number of levels of inclusion allowed:<br>• C90: 8<br>• C99 and later: 15 | |
| The number of members of a structure or union exceeds the limit specified in the standard. | Maximum number of members in a structure or union:<br>• C90: 127<br>• C99 and later: 1023 | |
| The number of levels of nesting in a structure exceeds the limit specified in the standard. | Maximum depth of nesting:<br>• C90: 15<br>• C99 and later: 63 | |
| The number of constants in a single enumeration exceeds the limit specified in the standard. | Maximum number of enumeration constants allowed:<br>• C90: 127<br>• C99 and later: 1023 | |
| An assembly language statement is used. | Checked for all C standards. | |
| A nonstandard preprocessor directive is used. | Checked for all C standards. | The rule checker flags uses of preprocessor directives that are not found in the C standard, for instance, `#ident`, `#alias` and `#assert`. |
| Unrecognized text follows a preprocessor directive. | Checked for all C standards. | The rule checker flags extraneous text following a preprocessor directive (line beginning with #). For instance:<br>`#include <header> code` |
| Unnamed unions or empty `structs` are used. | Checked for C90. | |
| An `enum` contains a trailing comma. | Checked for C90. | |

Standard compilation error messages do not lead to a violation of this MISRA rule.

**Tip** To mass-justify all results that come from the same cause, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Nonstandard C Syntax

```
#include <stdio.h>
#ident "@(#) Hello World"//noncompliant
extern int func(void);

void foo(void){
    int n = 2;
    asm ("leal (%0,%0,4),%0"
         : "=r" (n)
         : "0" (n));


    // standard inline assembly
    asm ("movq $60, %rax\n\t"
         "movq $2,  %rdi\n\t"
         "syscall");
}
```

The translation unit uses the nonstandard preprocessor directive `#ident`. Polyspace Flags the nonstandard syntax.

## Check Information

**Group:** Standard C Environment
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 1.2` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 1.2

Language extensions should not be used

## Description

### Rule Definition

*Language extensions should not be used*.

### Rationale

If a program uses language extensions, its portability is reduced. Even if you document the language extensions, the documentation might not describe the behavior in all circumstances.

### Polyspace Implementation

The rule checker flags these language extensions, depending on the version of the C standard used in the analysis. See `C standard version (-c-version)`.

- C90:
  - `long long int` type including constants
  - `long double` type
  - `inline` keyword
  - `_Bool` keyword
  - `short long int` type
  - Hexadecimal floating-point constants
  - Universal character names
  - Designated initializers
  - Local label declarations
  - `typeof` operator
  - Casts to union
  - Compound literals
  - Statements and declarations in expressions
  - `__func__` predefined identifier
  - `_Pragma` preprocessing operator
  - Macros with variable arguments list
  - `asm` functions
- C99:
  - `short long int` type
  - Local label declarations
  - `typeof` operator
  - Casts to union

- Statements and declarations in expressions
- `asm` functions

**Troubleshooting**

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Standard C Environment
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 1.1` | `Check MISRA C:2012 (-misra3)`

**Topics**

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 1.3

There shall be no occurrence of undefined or critical unspecified behaviour

## Description

### Rule Definition

*There shall be no occurrence of undefined or critical unspecified behaviour.*

### Rationale

C code that results in undefined or critical unspecified behavior might produce unexpected or incorrect results. Such code might behave differently in different implementations. Issues caused by undefined behavior in the code might be difficult to analyze because compilers might optimize the code assuming that undefined behavior does not occur.

**Note** Many MISRA C:2012 rules address specific undefined or critical unspecified behaviors. This rule applies to any undefined or critical unspecified behavior that is not addressed in any other rule.

### Polyspace Implementation

Polyspace flags these instances of undefined or critical undefined behavior:

- Use of `offsetof` on bit fields.
- Use of `offsetof` when the second argument is not a `struct` field of the first argument.
- Use of `defined` without an identifier.
- Use of an array of incomplete types.
- Use of a function like macros by using incorrect number of arguments.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Undefined Behaviors

```
#include <stddef.h>
static int bar = 0;
extern int bar;         /* Violation (8.8)*/

struct str {
  char a:8;
  char b[10];
  char c;
};
void foo() {
  offsetof(struct str, a);//Noncompliant
  offsetof(struct str, d);//Noncompliant
}
```

In this example, the function `foo` uses the macro `offsetof` on the bit field `str.a`. This behavior is undefined. Polyspace flags it. The function then calls `offsetof` on `str.d`. Because `d` is not a field of `str`, Polyspace flags it. These issues might cause compilation errors.

The variable `bar` is declared with both internal and external linkage. According to the C99 standard, declaring a variable to have both internal and external storage in the same file is undefined behavior. Polyspace flags this undefined behavior as a violation of rule 8.8. See `MISRA C:2012 Rule 8.8`.

## Check Information

**Group:** Standard C Environment
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Dir 4.1` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 1.4

Emergent language features shall not be used

## Description

### Rule Definition

*Emergent language features shall not be used*.

### Rationale

Some new language features in the C11 Standard have undefined, unspecified or implementation-defined behavior. These features might also exhibit well-defined behavior that defies developer expectations. Though rule 1.3 and directive 1.1 prohibits undefined and implementation-defined behavior, to avoid well-defined behavior that defies expectations, some language features are summarily discouraged using rule 1.4.

### Polyspace Implementation

The rule forbids use of the following language features:

- The `_Generic` operator.
- The `_Noreturn` function specifier and the `<stdnoreturn.h>` header file
- The `_Atomic` type specifier and the facilities provided by `<stdatomic.h>` (for instance, the macros beginning with `ATOMIC_` and functions beginning with `atomic_` implemented as macros in `<stdatomic.h>`).
- The `_Thread_local` storage class specifier and the facilities provided by `<threads.h>` (for instance, types such as `thrd_t` and functions such as `thrd_create`).
- The `_Alignas` alignment specifier, the `_Alignof` operator and the `<stdalign.h>` header file, and facilities therein (such as the `alignas` and `alignof` macros).
- All facilities in Annex K of the C11 Standard about 'Bound-checking interfaces', other than defining `__STDC_WANT_LIB_EXT1__` to '0'

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Facilities in Annex K of C11 Standard

```
#define __STDC_WANT_LIB_EXT1__ 1 //Noncompliant
#include <string.h>

void Copying_functions(void) {
    char buf1[10];
    char buf2[10];
    errno_t e;  //Noncompliant
    e = memcpy_s(buf1,sizeof(buf1),buf2,5); //Noncompliant
    e = memmove_s(buf1,sizeof(buf1),buf2,5); //Noncompliant
    e = strcpy_s(buf1,sizeof(buf1),buf2); //Noncompliant
    e = strncpy_s(buf1,sizeof(buf1),buf2,5); //Noncompliant
}
```

In this example, the macro `__STDC_WANT_LIB_EXT1__` is set to 1 so that the type `errno_t` as defined in the header `stdlib.h` can be used (in accordance with Annex K of the C11 Standard).

The checker flags both the setting of the macro to 1 and the definition of the `errno_t` variable, along with other functions from Annex K.

## Check Information

**Group:** Standard C Environment
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Dir 4.1` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 2.1

A project shall not contain unreachable code

## Description

### Rule Definition

*A project shall not contain unreachable code*.

### Rationale

Unless a program exhibits any undefined behavior, unreachable code cannot execute. The unreachable code cannot affect the program output. The presence of unreachable code can indicate an error in the program logic. Unreachable code that the compiler does not remove wastes resources, for example:

- It occupies space in the target machine memory.
- Its presence can cause a compiler to select longer, slower jump instructions when transferring control around the unreachable code.
- Within a loop, it can prevent the entire loop from residing in an instruction cache.

### Polyspace Implementation

Polyspace reports a defect if a statement in your code is not reachable.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Code Following `return` Statement

```
enum light { red, amber, red_amber, green };

enum light next_light ( enum light color )
{
    enum light res;

    switch ( color )
    {
    case red:
        res = red_amber;
        break;
    case red_amber:
        res = green;
        break;
    case green:
        res = amber;
        break;
    case amber:
        res = red;
        break;
    default:
    {
        error_handler ();
        break;
    }
    }

    res = color;
    return res;
    res = color;     /* Non-compliant */
}
```

In this example, the rule is violated because there is an unreachable operation following the `return` statement.

## Check Information

**Group:** Unused Code
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 14.3` | `MISRA C:2012 Rule 16.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 2.2

There shall be no dead code

## Description

### Rule Definition

*There shall be no dead code.*

### Rationale

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

### Polyspace Implementation

The rule checker reports rule violations on operations whose removal does not affect program behavior. The most common situation is an operation whose result is not used subsequently. For instance:

- You might perform an operation but not assign the result to a variable, even a temporary one.
- You might assign the result of an operation to a variable but not use the variable subsequently.
- You might assign a value to a variable and but not use the value subsequently. For more details, see `Write without a further read`.
- You might assign the result of an operation to a variable but immediately afterwards overwrite the variable value.

Other less common situations include calls to empty functions or redundant constant operations. Some of these operations involving constants occur in generated code and might be optimized away by your compiler. For instance, the operation `2u * 1u` is a redundant operation that a compiler is likely to optimize to `2u`. That is, the operation is unlikely to happen at run time. Irrespective of possible compiler optimizations, the rule checker reports a violation on the operation.

Polyspace does not flag an unused return of a non-`void` function because the function might have other side effects besides returning a value. Removing the function call might impact program behavior.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Redundant Operations

```
extern volatile unsigned int v;
extern char *p;
```

```
void f ( void ) {
    unsigned int x;


    ( void ) v;      /* Compliant - Exception*/
    ( int ) v;       /* Non-compliant  */
    v >> 3;          /* Non-compliant  */

    x = 3;           /* Non-compliant  */

    *p++;            /* Non-compliant  */
    ( *p )++;        /* Compliant  */
}
```

In this example, the rule is violated when an operation is performed on a variable, but the result of that operation is not used. For instance,

- The operations `(int)` and `>>` on the variable `v` are redundant because the results are not used.
- The operation `=` is redundant because the local variable `x` is not read after the operation.
- The operation `*` on `p++` is redundant because the result is not used.

The rule is not violated when:

- A variable is cast to `void`. The cast indicates that you are intentionally not using the value.
- The result of an operation is used. For instance, the operation `*` on `p` is not redundant, because `*p` is incremented.

### Redundant Function Call

```
void g ( void ) {
               /* Compliant  */
}

void h ( void) {
     g( );     /* Non-compliant */
}
```

In this example, `g` is an empty function. Though the function itself does not violate the rule, a call to the function violates the rule.

## Check Information

**Group:** Unused Code
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 17.7` | `Write without a further read` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 2.3

A project should not contain unused type declarations

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.

### Rule Definition

*A project should not contain unused type declarations*.

### Rationale

If a type is declared but not used, a reviewer does not know if the type is redundant or if it is unused by mistake.

### Additional Message in Report

A project should not contain unused type declarations: type XX is not used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unused Local Type

```
signed short unusedType (void){

    typedef signed short myType;   /* Non-compliant */
    return 67;

}

signed short usedType (void){

    typedef signed short myType;  /* Compliant */
    myType tempVar = 67;
    return tempVar;

}
```

In this example, in function `unusedType`, the `typedef` statement defines a new local type `myType`. However, this type is never used in the function. Therefore, the rule is violated.

The rule is not violated in the function `usedType` because the new type `myType` is used.

## Check Information

**Group:** Unused Code
**Category:** Advisory
**AGC Category:** Readability

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 2.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 2.4

A project should not contain unused tag declarations

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*A project should not contain unused tag declarations.*

### Rationale

If a tag is declared but not used, a reviewer does not know if the tag is redundant or if it is unused by mistake.

### Additional Message in Report

A project should not contain unused tag declarations: tag *`tag_name`* is not used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Tag Defined in Function but Not Used

```
void unusedTag ( void )
{
    enum state1 { S_init, S_run, S_sleep };    /* Non-compliant  */
}

void usedTag ( void )
{
    enum state2 { S_init, S_run, S_sleep };    /* Compliant  */
    enum state2 my_State = S_init;
}
```

In this example, in the function `unusedTag`, the tag `state1` is defined but not used. Therefore, the rule is violated.

### Tag Used in `typedef` Only

```
typedef struct record_t        /* Non-compliant  */
{
    unsigned short key;
    unsigned short val;
} record1_t;
```

```
typedef struct               /* Compliant */
{
    unsigned short key;
    unsigned short val;
} record2_t;

record1_t myRecord1_t;
record2_t myRecord2_t;
```

In this example, the tag `record_t` appears only in the `typedef` of `record1_t`. In the rest of the translation unit, the type `record1_t` is used. Therefore, the rule is violated.

## Check Information

**Group:** Unused Code
**Category:** Advisory
**AGC Category:** Readability

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 2.3` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 2.5

A project should not contain unused macro declarations

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*A project should not contain unused macro declarations.*

### Rationale

If a macro is declared but not used, a reviewer does not know if the macro is redundant or if it is unused by mistake.

### Additional Message in Report

A project should not contain unused macro declarations: macro `macro_name` is not used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Unused Macro Definition

```
void use_macro (void)
{
    #define SIZE 4
    #define DATA 3//Noncompliant

    use_int16(SIZE);
}
```

In this example, the macro `DATA` is never used in the `use_macro` function. Polyspace flags the unused macro.

## Check Information

**Group:** Unused Code
**Category:** Advisory
**AGC Category:** Readability

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 2.6

A function should not contain unused label declarations

## Description

### Rule Definition

*A function should not contain unused label declarations*.

### Rationale

If you declare a label but do not use it, it is not clear to a reviewer of your code if the label is redundant or unused by mistake.

### Additional Message in Report

A function should not contain unused label declarations.

Label *`label_name`* is not used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Unused Label Declarations

```
void use_var(signed short);

void unused_label ( void )
{
    signed short x = 6;

label1:                         /* Non-compliant - label1 not used */
    use_var ( x );
}

void used_label ( void )
{
    signed short x = 6;

    for (int i=0; i < 5; i++) {
        if ( i==2 ) goto label1;
    }

label1:                        /* Compliant - label1 used */
    use_var ( x );
}
```

In this example, the rule is violated when the label `label1` in function `unused_label` is not used.

## Check Information

**Group:** Unused code
**Category:** Advisory
**AGC Category:** Readability

## Version History

**Introduced in R2015b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 2.7

There should be no unused parameters in functions

## Description

### Rule Definition

*There should be no unused parameters in functions.*

### Rationale

If a parameter is unused, it is possible that the implementation of the function does not match its specifications. This rule can highlight such mismatches.

### Additional Message in Report

There should be no unused parameters in functions.

Parameter *`parameter_name`* is not used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unused Function Parameters

```
double func(int param1, int* param2) { /* Non-compliant */
    return (param1/2.0);
}
```

In this example, the rule is violated because the parameter `param2` is not used.

## Check Information

**Group:** Unused code<br>
**Category:** Advisory<br>
**AGC Category:** Readability

# Version History

**Introduced in R2015b**

## See Also

`Unused parameter` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”

“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 3.1

The character sequences /* and // shall not be used within a comment

## Description

### Rule Definition

*The character sequences /* and // shall not be used within a comment.*

### Rationale

These character sequences are not allowed in code comments because:

- If your code contains a /* or a // in a /* */ comment, it typically means that you have inadvertently commented out code.
- If your code contains a /* in a // comment, it typically means that you have inadvertently uncommented a /* */ comment.

### Polyspace Implementation

You cannot annotate this rule in the source code.

For information on annotations, see “Annotate Code and Hide Known or Acceptable Results”.

### Additional Message in Report

The character sequence /* shall not appear within a comment.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### /* Used in // Comments

```
int x;
int y;
int z;

void non_compliant_comments ( void )
{
    x = y //      /* Non-compliant
        + z
        //  */
        ;
    z++;    //    Compliant with exception: // permitted within a // comment
}

void compliant_comments ( void )
{
    x = y /*      Compliant
      + z
      */
        ;
    z++;    //    Compliant with exception: // is permitted within a // comment
}
```

In this example, in the `non_compliant_comments` function, the `/*` character occurs in what appears to be a `//` comment, violating the rule. Because of the comment structure, the operation that takes place is `x = y + z;`. However, without the two `//`-s, an entirely different operation `x=y;` takes place. It is not clear which operation is intended.

Use a comment format that makes your intention clear. For instance, in the `compliant_comments` function, it is clear that the operation `x=y;` is intended.

## Check Information

**Group:** Comments
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 3.2

Line-splicing shall not be used in `//` comments

## Description

### Rule Definition

*Line-splicing shall not be used in // comments.*

### Rationale

Line-splicing occurs when the `\` character is immediately followed by a new-line character. Line splicing is used for statements that span multiple lines.

If you use line-splicing in a `//` comment, the following line can become part of the comment. In most cases, the `\` is spurious and can cause unintentional commenting out of code.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Line Splicing in // Comment

```
#include <stdbool.h>

extern _Bool b;

void func ( void )
{
    unsigned short x = 0;   // Non-compliant - Line-splicing \
    if ( b )
    {
        ++b;
    }
}
```

Because of line-splicing, the statement `if ( b )` is a part of the previous `//` comment. Therefore, the statement `b++` always executes, making the `if` block redundant.

## Check Information

**Group:** Comments
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C:2012 Rule 4.1

Octal and hexadecimal escape sequences shall be terminated

## Description

### Rule Definition

*Octal and hexadecimal escape sequences shall be terminated*.

### Rationale

There is potential for confusion if an octal or hexadecimal escape sequence is followed by other characters. For example, the character constant `'\x1f'` consists of a single character, whereas the character constant `'\x1g'` consists of the two characters `'\x1'` and `'g'`. The manner in which multi-character constants are represented as integers is implementation-defined.

If every octal or hexadecimal escape sequence in a character constant or string literal is terminated, you reduce potential confusion.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Compliant and Noncompliant Escape Sequences

```
const char *s1 = "\x41g";     /* Non-compliant */
const char *s2 = "\x41" "g";  /* Compliant - Terminated by end of literal */
const char *s3 = "\x41\x67";  /* Compliant - Terminated by another escape sequence*/

int c1 = '\141t';             /* Non-compliant */
int c2 = '\141\t';            /* Compliant - Terminated by another escape sequence*/
```

In this example, the rule is violated when an escape sequence is not terminated with the end of string literal or another escape sequence.

## Check Information

**Group:** Character Sets and Lexical Conventions
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

**Topics**

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 4.2

Trigraphs should not be used

## Description

### Rule Definition

*Trigraphs should not be used*.

### Rationale

You denote trigraphs with two question marks followed by a specific third character (for instance,`'??-'` represents a `'~'` (tilde) character and `'??)'` represents a `']'`). These trigraphs can cause accidental confusion with other uses of two question marks.

**Note** Digraphs (`<:  :>`, `<%  %>`, `%:`, `%:%:`) are permitted because they are tokens.

### Polyspace Implementation

The Polyspace analysis converts trigraphs to the equivalent character for the defect analysis. However, Polyspace also raises a MISRA violation.

The standard requires that trigraphs must be transformed *before* comments are removed during preprocessing. Therefore, Polyspace raises a violation of this rule even if a trigraph appears in code comments.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Character Sets and Lexical Conventions
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 5.1

External identifiers shall be distinct

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

### Rule Definition

*External identifiers shall be distinct.*

### Rationale

External identifiers are ones declared with global scope or storage class `extern`.

If the two long names are almost identical and differ only in the later part, they can be easily mistaken for each other. The readability of the code is reduced.

### Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 31 characters. For C90, the difference must occur between the first 6 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`. You can change the number of characters compared using the option `-code-behavior-specifications`. See `-code-behavior-specifications`.

### Additional Message in Report

External `%s %s` conflicts with the external identifier XX in file YY.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### C90: First Six Characters of Identifiers Not Unique

```
int engine_temperature_raw;
int engine_temperature_scaled;   /* Non-compliant */
int engin2_temperature;          /* Compliant */
```

In this example, the identifier `engine_temperature_scaled` has the same first six characters as a previous identifier, `engine_temperature_raw`.

### C99: First 31 Characters of Identifiers Not Unique

```
int engine_exhaust_gas_temperature_raw;
int engine_exhaust_gas_temperature_scaled; /* Non-compliant */

int eng_exhaust_gas_temp_raw;
int eng_exhaust_gas_temp_scaled;           /* Compliant */
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same first 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

#### C90: First Six Characters Identifiers in Different Translation Units Differ in Case Alone

```
/* file1.c */
int abc = 0;


/* file2.c */
int ABC = 0; /* Non-compliant */
```

In this example, the implementation supports 6 significant case-insensitive characters in *external identifiers*. The identifiers in the two translation are different but are not distinct in their significant characters.

## Check Information

**Group:** Identifiers
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.4` | `MISRA C:2012 Rule 5.5` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 5.2

Identifiers declared in the same scope and name space shall be distinct

## Description

### Rule Definition

*Identifiers declared in the same scope and name space shall be distinct.*

### Rationale

If the two long names are almost identical and differ only in the later part, they can be easily mistaken for each other. The readability of the code is reduced.

### Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 63 characters. In C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`. You can change the number of characters compared using the option `-code-behavior-specifications`. See `-code-behavior-specifications`.

### Additional Message in Report

Identifier XX has same significant characters as identifier YY.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### C90: First 31 Characters of Identifiers Not Unique

```
extern int engine_exhaust_gas_temperature_raw;
static int engine_exhaust_gas_temperature_scaled;      /* Non-compliant */

extern double raw_engine_exhaust_gas_temperature;
static double scaled_engine_exhaust_gas_temperature;  /* Compliant */

void func ( void )
{
  /* Not in the same scope */
  int engine_exhaust_gas_temperature_local;            /* Compliant */
}
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

The rule does not apply if the two identifiers have the same 31 characters but have different scopes. For instance, `engine_exhaust_gas_temperature_local` has the same 31 characters as `engine_exhaust_gas_temperature_raw` but different scope.

### Check Information

**Group:** Identifiers
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

### See Also

`MISRA C:2012 Rule 5.1` | `MISRA C:2012 Rule 5.3` | `MISRA C:2012 Rule 5.4` | `MISRA C:2012 Rule 5.5` | `Check MISRA C:2012 (-misra3)`

#### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 5.3

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

## Description

### Rule Definition

*An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.*

### Rationale

If two identifiers have the same name but different scope, the identifier in the inner scope hides the identifier in the outer scope. All uses of the identifier name refers to the identifier in the inner scope. This behavior forces the developer to keep track of the scope and reduces code readability.

### Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 63 characters. In C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`. You can change the number of characters compared using the option `-code-behavior-specifications`. See `-code-behavior-specifications`.

If the identifier that is hidden is declared in a Standard Library header and you do not provide the header for the analysis, the issue is not shown. To see potential conflicts with identifiers declared in a Standard Library header, provide your compiler implementation of the headers for the Polyspace analysis. See “Provide Standard Library Headers for Polyspace Analysis”.

### Additional Message in Report

Variable XX hides variable XX (FILE line LINE column COLUMN).

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Local Variable Hidden by Another Local Variable in Inner Block

```
typedef signed short int16_t;

void func( void )
{
    int16_t i;
    {
        int16_t i;                /* Non-compliant */
        i = 3;
    }
}
```

In this example, the identifier `i` defined in the inner block in `func` hides the identifier `i` with function scope.

It is not immediately clear to a reader which `i` is referred to in the statement `i=3`.

#### Global Variable Hidden by Function Parameter

```
typedef signed short int16_t;

struct astruct
{
    int16_t m;
};

extern void g ( struct astruct *p );
int16_t xyz = 0;

void func ( struct astruct xyz )  /* Non-compliant */
{
    g ( &xyz );
}
```

In this example, the parameter `xyz` of function `func` hides the global variable `xyz`.

It is not immediately clear to a reader which `xyz` is referred to in the statement `g (&xyz )`.

## Check Information

**Group:** Identifiers
**Category:** Required
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.8` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 5.4

Macro identifiers shall be distinct

## Description

### Rule Definition

*Macro identifiers shall be distinct.*

### Rationale

The names of macro identifiers must be distinct from both other macro identifiers and their parameters.

### Polyspace Implementation

The checker raises a violation if two macros that have the same first 63 characters are defined with different values. The checker does not raise a violation if:

- Two macros with the same first 63 characters are defined with the same value (even an empty value).
- The same macro is defined with different values but the macro is undefined in between.

The cutoff of 63 characters applies to a C99-based analysis. In C90, the cutoff is 31 characters. In other words, the checker considers two macros as effectively the same if there is no difference in their first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`. You can change the number of characters compared using the option `-code-behavior-specifications`. See `-code-behavior-specifications`.

### Additional Message in Report

- Macro identifiers shall be distinct. Macro XX has same significant characters as macro YY.
- Macro identifiers shall be distinct. Macro parameter XX has same significant characters as macro parameter YY in macro ZZ.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### C90: First 31 Characters of Macro Names Not Unique

```
#define engine_exhaust_gas_temperature_raw egt_r
#define engine_exhaust_gas_temperature_scaled egt_s   /* Non-compliant */

#define engine_exhaust_gas_temp_raw egt_r
#define engine_exhaust_gas_temp_scaled egt_s          /* Compliant */
```

In this example, the macro `engine_exhaust_gas_temperature_scaled egt_s` has the same first 31 characters as a previous macro `engine_exhaust_gas_temperature_scaled`.

## Check Information

**Group:** Identifiers
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 5.1` | `MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.5` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 5.5

Identifiers shall be distinct from macro names

## Description

### Rule Definition

*Identifiers shall be distinct from macro names.*

### Rationale

The rule requires that macro names that exist only prior to processing must be different from identifier names that also exist after preprocessing. Keeping macro names and identifiers distinct help avoid confusion.

### Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 63 characters. In C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`. You can change the number of characters compared using the option `-code-behavior-specifications`. See `-code-behavior-specifications`.

### Additional Message in Report

Identifier XX has same significant characters as macro YY.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Macro Names Same as Identifier Names

```
#define Sum_1(x, y) ( ( x ) + ( y ) )
short Sum_1;                    /* Non-compliant */

#define Sum_2(x, y) ( ( x ) + ( y ) )
short x = Sum_2 ( 1, 2 );       /* Compliant */
```

In this example, `Sum_1` is both the name of an identifier and a macro. `Sum_2` is used only as a macro.

### C90: First 31 Characters of Macro Name Same as Identifier Name

```
#define low_pressure_turbine_temperature_1 lp_tb_temp_1
static int low_pressure_turbine_temperature_2;   /* Non-compliant  */
```

In this example, the identifier `low_pressure_turbine_temperature_2` has the same first 31 characters as a previous macro `low_pressure_turbine_temperature_1`.

## Check Information

**Group:** Identifiers
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 5.1` | `MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 5.6

A typedef name shall be a unique identifier

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*A typedef name shall be a unique identifier.*

### Rationale

Reusing a `typedef` name as another `typedef` or as the name of a function, object or `enum` constant can cause developer confusion.

### Additional Message in Report

XX conflicts with the typedef name YY.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### typedef Names Reused

```
void func ( void ){
  {
    typedef unsigned char u8_t;
  }
  {
    typedef unsigned char u8_t; /* Non-compliant */
  }
}

typedef float mass;
void func1 ( void ){
  float mass = 0.0f;            /* Non-compliant */
}
```

In this example, the `typedef` name `u8_t` is used twice. The `typedef` name `mass` is also used as an identifier name.

### typedef Name Same as Structure Name

```
typedef struct list{          /* Compliant - exception */
  struct list *next;
  unsigned short element;
} list;
```

```
typedef struct{
  struct chain{               /* Non-compliant */
    struct chain *list2;
    unsigned short element;
  } s1;
  unsigned short length;
} chain;
```

In this example, the `typedef` name `list` is the same as the original name of the `struct` type. The rule allows this exceptional case.

However, the `typedef` name `chain` is not the same as the original name of the `struct` type. The name `chain` is associated with a different `struct` type. Therefore, it clashes with the `typedef` name.

## Check Information

**Group:** Identifiers
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 5.7` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 5.7

A tag name shall be a unique identifier

## Description

### Rule Definition

*A tag name shall be a unique identifier*.

### Rationale

Reusing a tag name can cause developer confusion.

### Additional Message in Report

XX conflicts with the tag name YY.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Identifiers
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 5.6` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 5.8

Identifiers that define objects or functions with external linkage shall be unique

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*Identifiers that define objects or functions with external linkage shall be unique.*

### Rationale

External identifiers are those declared with global scope or with storage class `extern`. Reusing an external identifier name can cause developer confusion.

Identifiers defined within a function have smaller scope. Even if names of such identifiers are not unique, they are not likely to cause confusion.

### Additional Message in Report

- Object XX conflicts with the object name YY.
- Function XX conflicts with the function name YY.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Identifiers
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 5.3` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 5.9

Identifiers that define objects or functions with internal linkage should be unique

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.

### Rule Definition

*Identifiers that define objects or functions with internal linkage should be unique*.

### Rationale

Identifiers that have internal linkage are accessible only in the translation unit where they are declared. These identifiers are typically declared as `static`. If such identifiers are nonunique, the code might become difficult to understand and lead to unexpected results.

### Polyspace Implementation

Polyspace flags the `static` variable names that are nonunique within the same translation unit.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Nonunique Identifiers

```
#include<stdint.h>
#include <assert.h>
static int testGlobal;
void foo(){
static char testGlobal;//Noncompliant
if(1){
    static char testGlobal;//Noncompliant
}
}
```

In this example, the identifier `testGlobal` is used for declaring three variables in three different scopes. Because the identifiers are `static` and share a nonunique name, Polyspace flags the repetitions of the identifier.

## Check Information

**Group:** Identifiers
**Category:** Advisory
**AGC Category:** Readability

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 8.10` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 6.1

Bit-fields shall only be declared with an appropriate type

## Description

### Rule Definition

*Bit-fields shall only be declared with an appropriate type*.

### Rationale

Using `int` for a bit-field type is implementation-defined because bit-fields of type `int` can be either `signed` or `unsigned`.

The use of `enum`, `short char`, or any other type of bit-field is not permitted in C90 because the behavior is undefined.

In C99, the implementation can potentially define other integer types that are permitted in bit-field declarations.

### Polyspace Implementation

The checker flags data types for bit-fields other than these allowed types:

- C90: `signed int` or `unsigned int` (or `typedef`-s that resolve to these types)
- C99: `signed int`, `unsigned int` or `_Bool` (or `typedef`-s that resolve to these types)

The results depend on the version of the C standard used in the analysis. See `C standard version (-c-version)`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Types
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”

“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 6.2

Single-bit named bit fields shall not be of a signed type

## Description

### Rule Definition

*Single-bit named bit fields shall not be of a signed type*.

### Rationale

According to the C99 Standard Section 6.2.6.2, a single-bit signed bit-field has one sign bit and no value bits. In any representation of integers, zero value bits cannot specify a meaningful value.

A single-bit signed bit-field is therefore unlikely to behave in a useful way. Its presence is likely to indicate programmer confusion.

Although the C90 Standard does not provide much detail regarding the representation of types, the same single-bit bit-field considerations apply.

### Polyspace Implementation

This rule does not apply to unnamed bit fields because their values cannot be accessed.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Types
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 7.1

Octal constants shall not be used

## Description

### Rule Definition

*Octal constants shall not be used*.

### Rationale

Octal constants are denoted by a leading zero. Developers can mistake an octal constant as a decimal constant with a redundant leading zero.

### Polyspace Implementation

If you use octal constants in a macro definition, the rule checker flags the issue even if the macro is not used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

#### Example - Use of octal constants

```
#define CST     021             /* Noncompliant */
#define VALUE   010             /* Noncompliant */
#if 010 == 01                   /* Noncompliant*/
#define CST 021                 /* Noncompliant */
#endif

extern short code[5];
static char* str2 = "abcd\0efg";  /* Compliant */

void main(void) {
    int value1 = 0;               /* Compliant */
    int value2 = 01;              /* Noncompliant*/
    int value3 = 1;               /* Compliant */
    int value4 = '\109';          /* Compliant */

    code[1] = 109;                /* Compliant     - decimal 109 */
    code[2] = 100;                /* Compliant     - decimal 100 */
    code[3] = 052;                /* Noncompliant */
    code[4] = 071;                /* Noncompliant */

    if (value1 != CST) {
        value1 = !(value1 != 0);  /* Compliant */
    }
}
```

In this example, Polyspace flags the use of octal constants.

## Check Information

**Group:** Literals and Constants
**Category:** Required
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 7.2

A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type

## Description

### Rule Definition

*A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type*.

### Rationale

The signedness of a constant is determined from:

- Value of the constant.
- Base of the constant: octal, decimal or hexadecimal.
- Size of the various types.
- Any suffixes used.

Unless you use a suffix `u` or `U`, another developer looking at your code cannot determine easily whether a constant is signed or unsigned.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use Suffix to Specify Unsigned Type

```
const unsigned long C[] = {
    0x9421FFD0, /*Noncompliant*/
    0x5322E762,
    0x80000000, /*Noncompliant*/
    0x7FFFFFFF,
    0x00000001,
    0x83241947, /*Noncompliant*/
    0x57112957,
    0x2640EA23
};

const unsigned long D[] = {
    0x9421FFD0U, /*Compliant*/
    0x80000000U, /*Compliant*/
    0x83241947U, /*Compliant*/
};
```

In this example, Polyspace flags the unsigned members of `C`. For instance, `0x9421FFD0` is an unsigned number in a 32 bit environment because it exceeds the capacity of a signed integer. Because the unsigned number lacks the suffix `u` or `U`, Polyspace flags it. In `D`, the unsigned numbers use the suffix and are not flagged.

## Check Information

**Group:** Literals and Constants
**Category:** Required
**AGC Category:** Readability

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 7.3

The lowercase character “l” shall not be used in a literal suffix

## Description

### Rule Definition

*The lowercase character “l” shall not be used in a literal suffix.*

### Rationale

The lowercase character “l” can be confused with the digit “1”. Use the uppercase “L” instead.

### Polyspace Implementation

Polyspace reports a violation if you use the lowercase character "l" in a literal suffix. Violations of this rule are not reported on unused macros.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Lowercase "l" in Literal Suffix

```
#define PI 3.14159l// Compliant- Not flagged because
                    // the macro is unused
#define EULERNUM 2.71828l//Noncompliant- Flagged because
                         // macro is used
void func(long);
void foo(void){
    long a = 10l;//Noncompliant
    long b = 10L;//Compliant
    long c = 10lL;//Noncompliant
    func(EULERNUM);
}
```

In this example, Polyspace flags the literals that have a lowercase "l" in their suffix. Polyspace does not flag unused macros that have "l" in their suffix.

## Check Information

**Group:** Literals and Constants
**Category:** Required
**AGC Category:** Readability

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 7.4

A string literal shall not be assigned to an object unless the object’s type is “pointer to const-qualified char”

## Description

### Rule Definition

*A string literal shall not be assigned to an object unless the object’s type is “pointer to const-qualified char”.*

### Rationale

This rule prevents assignments that allow modification of a string literal.

An attempt to modify a string literal can result in undefined behavior. For example, some implementations can store string literals in read-only memory. An attempt to modify the string literal can result in an exception or crash.

### Polyspace Implementation

The rule checker flags assignment of string literals to:

- Pointers with data type other than `const char*`.
- Arrays with data type other than `const char`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Incorrect Assignment of String Literal

```
char *str1 = "xxxxxx";            // Non-Compliant
const char *str2 = "xxxxxx";      // Compliant

void checkSystem1(char*);
void checkSystem2(const char*);

void main() {
 checkSystem1("xxxxxx");    // Non-Compliant
 checkSystem2("xxxxxx");    // Compliant
}
```

In this example, the rule is not violated when string literals are assigned to `const char*` pointers, either directly or through copy of function arguments. The rule is violated only when the `const` qualifier is not used.

## Check Information

**Group:** Literals and Constants
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 11.4` | `MISRA C:2012 Rule 11.8` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 8.1

Types shall be explicitly specified

## Description

### Rule Definition

*Types shall be explicitly specified*.

### Rationale

In some circumstances, you can omit types from the C90 standard. In those cases, the `int` type is implicitly specified. However, the omission of an explicit type can lead to confusion. For example, in the declaration `extern void foo (char c, const k);`, the type of `k` is `const int`, but you might expect `const char`.

You might be using an implicit type in:

- Object declarations
- Parameter declarations
- Member declarations
- `typedef` declarations
- Function return types

### Polyspace Implementation

The rule checker flags situations where a function parameter or return type is not explicitly specified.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Implicit Types

```
static foo(int a);  /* Non compliant */
static void bar(void);      /* Compliant */
```

In this example, the rule is violated because the return type of `foo` is implicit.

## Check Information

**Group:** Declarations and Definitions
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 8.2` | `Check MISRA C:2012 (-misra3)`

### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 8.2

Function types shall be in prototype form with named parameters

## Description

### Rule Definition

*Function types shall be in prototype form with named parameters.*

### Rationale

The rule requires that you specify names and data types for all the parameters in a declaration. The parameter names provide useful information regarding the function interface. A mismatch between a declaration and definition can indicate a programming error. For instance, you mixed up parameters when defining the function. By insisting on parameter names, the rule allows a code reviewer to detect this mismatch.

### Polyspace Implementation

The rule checker shows a violation if the parameters in a function declaration or definition are missing names or data types.

### Additional Message in Report

- Too many arguments to `function_name`.
- Too few arguments to `function_name`.
- Function types shall be in prototype form with named parameters.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Function Prototype Without Named Parameters

```
extern int func(int);   /* Non compliant */
extern int func2(int n);   /* Compliant */

extern int func3();   /* Non compliant */
extern int func4(void);   /* Compliant */
```

In this example, the declarations of `func` and `func3` are noncompliant because the parameters are missing or do not have names.

## Check Information

**Group:** Declarations and Definitions
**Category:** Required

**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 8.1` | `MISRA C:2012 Rule 8.4` | `MISRA C:2012 Rule 17.3` | `Check MISRA C:2012 (-misra3)`

### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 8.3

All declarations of an object or function shall use the same names and type qualifiers

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*All declarations of an object or function shall use the same names and type qualifiers*.

### Rationale

Consistently using parameter names and types across declarations of the same object or function encourages stronger typing. It is easier to check that the same function interface is used across all declarations.

### Polyspace Implementation

The rule checker detects situations where parameter names or data types are different between multiple declarations or the declaration and the definition. The checker considers declarations in all translation units and flags issues that are not likely to be detected by a compiler.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Mismatch in Parameter Names

```
extern int div (int num, int den);

int div(int den, int num) { /* Non compliant */
    return(num/den);
}
```

In this example, the rule is violated because the parameter names in the declaration and definition are switched.

### Mismatch in Parameter Data Types

```
typedef unsigned short width;
typedef unsigned short height;
typedef unsigned int area;

extern area calculate(width w, height h);

area calculate(width w, width h) { /* Non compliant */
    return w*h;
}
```

In this example, the rule is violated because the second argument of the `calculate` function has data type:

- `height` in the declaration.
- `width` in the definition.

The rule is violated even though the underlying type of `height` and `width` are identical.

## Check Information

**Group:** Declarations and Definitions
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 8.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 8.4

A compatible declaration shall be visible when an object or function with external linkage is defined

## Description

### Rule Definition

*A compatible declaration shall be visible when an object or function with external linkage is defined.*

### Rationale

If a declaration is visible when an object or function is defined, it allows the compiler to check that the declaration and the definition are compatible.

This rule with `MISRA C:2012 Rule 8.5` enforces the practice of declaring an object (or function) in a header file and including the header file in source files that define or use the object (or function).

### Polyspace Implementation

The rule checker detects situations where:

- An object or function is defined without a previous declaration.
- There is a data type mismatch between the object or function declaration and definition. Such a mismatch also causes a compilation error.

The checker now flags tentative definitions (variables declared without an `extern` specifier and not explicitly defined). To avoid the rule violation, declare the variable `static` (defined in one file only), or declare the variable `extern` and follow the declaration with a definition.

### Additional Message in Report

- Global definition of *`variable_name`* variable has no previous declaration.
- Function *`function_name`* has no visible compatible prototype at definition.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Definition Without Previous Declaration

Header file:

```
/* file.h */
extern int var2;
void func2(void);
```

Source file:

```
/* file.c */
#include "file.h"

int var1 = 0;    /* Non compliant */
int var2 = 0;    /* Compliant */

void func1(void) {   /* Non compliant */
}

void func2(void) {   /* Compliant */
}
```

In this example, the definitions of `var1` and `func1` are noncompliant because they are not preceded by declarations.

#### Mismatch in Parameter Data Types

```
void func(int param1, int param2);

void func(int param1, unsigned int param2) { /* Non compliant */
}
```

In this example, the definition of `func` has a different parameter type from its declaration. The declaration mismatch might cause a compilation error. Polyspace flags the mismatch.

## Check Information

**Group:** Declarations and Definitions
**Category:** Required
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 8.2` | `MISRA C:2012 Rule 8.3` | `MISRA C:2012 Rule 8.5` | `MISRA C:2012 Rule 17.3` | `Check MISRA C:2012 (-misra3)`

#### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 8.5

An external object or function shall be declared once in one and only one file

## Description

### Rule Definition

*An external object or function shall be declared once in one and only one file.*

### Rationale

Declaring an identifier in a header file enables you to include the header file in any translation unit where the identifier is defined or used. Modularizing the declarations in header files helps maintain consistency between:

- The declaration and the definition.
- The declarations in different translation units.

The rule enforces the practice of declaring external objects or functions in header files.

### Polyspace Implementation

The rule checker checks explicit and implicit `extern` declarations (tentative definitions are ignored). The checker flags variables or functions:

- Declared `extern` in a nonheader file
- Declared multiple times, for instance, once in a header and once in a nonheader file.

This checker:

- Ignores compiler-specific, nonportable ways of overriding function declarations, such as `pragma weak` or `__attribute__((weak))`.
- Ignores typedefs and function declarations that are not used in the code.
- Considers redeclaration as a rule violation. For instance, if you declare a weak symbol in your code, and then redeclare the symbol later, Polyspace reports a violation. If you do not want to fix such violations, add a comment to your result or code to avoid another review. See:
  - "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
  - "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Extern Declaration in Nonheader File

In this example, the declaration of external function `func2()` is noncompliant because it occurs in a nonheader file. The other external object and function declarations occur in a header file and comply with this rule. To fix this issue, declare `func2()` in a header file.

Header file:

```
/* header.h */
extern int var;
extern void func1(void);   /* Compliant */
```

Source file:

```
/* module.c */
#include "header.h"

extern void func2(void);   /* Noncompliant */

/* Definitions */
int var = 0;
void func1(void) {}
```

### Violation Caused by Explicit Declaration and Implicit Declaration

In this example, the function `func()` is declared in `header1.h`. In `header2.h`, the function is called. In `module2.c`, the file `header2.h` is included, but `header1.h` is not included, perhaps inadvertently. Because `module2.c` includes a call to `func()` but no declaration, the compiler generates an implicit declaration for the function during compilation. The file `module1.c` includes `header1.h`, which contains an explicit declaration. Having an explicit and an implicit declaration for the same external function is noncompliant with this rule. Polyspace raises a violation. Polyspace reports a defect. In the **Result Details** section, Polyspace flags :

- The explicit declaration in `header1.h`
- The function call in `module2.c`, which triggers the creation of the implicit declaration.

```
//header1.h

extern void func(void);//Noncompliant
```

```
//header2.h

#define __STATIC_INLINE static inline

__STATIC_INLINE void Reset(void)
{
    //...
    func();
}
```

```
//module1.c
#include "header2.h"
#include "header1.h"

void foo(void)
{
}
```

```
//module2.c
#include "header2.h"

void bar(void)
{
}
```

To fix this violation, include the file `header1.h` in `module2.c`.

### Violation Caused by Implicit Declaration

In this example, the declaration of the function `func()` is in the header `header.h`. The source files `module1.c` and `module2.c` contains calls to `func()`, but the header file is not included in these source files, perhaps by mistake. As a result, the compiler creates two implicit declaration of `func()`, causing a violation of this rule. Polyspace reports the violation. In the **Result Details** section, Polyspace flags the function calls that trigger the creation of the implicit declarations.

```
//header.h

extern void func(void);
```

```
//module1.c

void foo(void)
{
    func();
}
```

```
//module2.c

void bar(void)
{
    func(); //Noncompliant
}
```

To fix this violation, include the file `header.h` in `module1.c` and `module2.c`.

## Check Information

**Group:** Declarations and Definitions
**Category:** Required
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 8.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 8.6

An identifier with external linkage shall have exactly one external definition

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*An identifier with external linkage shall have exactly one external definition.*

### Rationale

If you use an identifier for which multiple definitions exist in different files or no definition exists, the behavior is undefined.

Multiple definitions in different files are not permitted by this rule even if the definitions are the same.

### Polyspace Implementation

The checker flags multiple definitions only if the definitions occur in different files.

The checker does not consider tentative definitions as definitions. For instance, the following code does not violate the rule:

```
int val;
int val=1;
```

The checker does not show a violation if a function is not defined at all but declared with external linkage and called in the source code.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Variable Multiply Defined

First source file:

```
/* file1.c */
extern int var = 1;
```

Second source file:

```
/* file2.c */
int var = 0;   /* Non compliant */
```

In this example, the global variable `var` is multiply defined. Unless explicitly specified with the `static` qualifier, the variables have external linkage.

### Function Multiply Defined

Header file:

```
/* file.h */
int func(int param);
```

First source file:

```
/* file1.c */
#include "file.h"

int func(int param) {
    return param+1;
}
```

Second source file:

```
/* file2.c */
#include "file.h"

int func(int param) {  /* Non compliant */
    return param-1;
}
```

In this example, the function `func` is multiply defined. Unless explicitly specified with the `static` qualifier, the functions have external linkage.

## Check Information

**Group:** Declarations and Definitions
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 8.7

Functions and objects should not be defined with external linkage if they are referenced in only one translation unit

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.

### Rule Definition

*Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.*

### Rationale

Compliance with this rule avoids confusion between your identifier and an identical identifier in another translation unit or library. If you restrict or reduce the visibility of an object by giving it internal linkage or no linkage, you or someone else is less likely to access the object inadvertently.

### Polyspace Implementation

The rule checker flags:

- Objects that are defined at file scope without the `static` specifier but used only in one file.
- Functions that are defined without the `static` specifier but called only in one file.

If you intend to use the object or function in one file only, declare it static.

If your code does not contain a `main` function and you use options such as `Variables to initialize (-main-generator-writes-variables)` with value `custom` to explicitly specify a set of variables to initialize, the checker does not flag those variables. The checker assumes that in a real application, the file containing the `main` must initialize the variables in addition to any file that currently uses them. Therefore, the variables are used in more than one translation unit.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Variable with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
```

First source file:

```
/* file1.c */
#include "file.h"

int var;    /* Compliant */
int var2;   /* Non compliant */
static int var3; /* Compliant */

void reset(void);

void reset(void) {
    var = 0;
    var2 = 0;
    var3 = 0;
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment(int var2);

void increment(int var2) {
    var++;
    var2++;
}
```

In this example:

- The declaration of `var` is compliant because `var` is declared with external linkage and used in multiple files.
- The declaration of `var2` is noncompliant because `var2` is declared with external linkage but used in one file only.

  It might appear that `var2` is defined in both files. However, in the second file, `var2` is a parameter with no linkage and is not the same as the `var2` in the first file.
- The declaration of `var3` is compliant because `var3` is declared with internal linkage (with the `static` specifier) and used in one file only.

### Function with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
extern void increment1 (void);
```

First source file:

```
/* file1.c */
#include "file.h"

int var;
```

```
void increment2(void);
static void increment3(void);
void func(void);

void increment2(void) { /* Non compliant */
    var+=2;
}

static void increment3(void) { /* Compliant */
    var+=3;
}

void func(void) {
    increment1();
    increment2();
    increment3();
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment1(void) { /* Compliant */
    var++;
}
```

In this example:

- The definition of `increment1` is compliant because `increment1` is defined with external linkage and called in a different file.
- The declaration of `increment2` is noncompliant because `increment2` is defined with external linkage but called in the same file and nowhere else.
- The declaration of `increment3` is compliant because `increment3` is defined with internal linkage (with the `static` specifier) and called in the same file and nowhere else.

## Check Information

**Group:** Declarations and Definitions
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"

“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 8.8

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage

## Description

### Rule Definition

*The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.*

### Rationale

If you do not use the `static` specifier consistently in all declarations of objects with internal linkage, you might declare the same object with external and internal linkage.

In this situation, the linkage follows the earlier specification that is visible (C99 Standard, Section 6.2.2). For instance, if the earlier specification indicates internal linkage, the object has internal linkage even though the latter specification indicates external linkage. If you notice the latter specification alone, you might expect otherwise.

### Polyspace Implementation

The rule checker detects situations where:

- The same object is declared multiple times with different storage specifiers.
- The same function is declared and defined with different storage specifiers.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Linkage Conflict Between Variable Declarations

```
static int foo = 0;
extern int foo;         /* Non-compliant */

extern int hhh;
static int hhh;         /* Non-compliant */
```

In this example, the first line defines `foo` with internal linkage. The first line is compliant because the example uses the `static` keyword. The second line does not use `static` in the declaration, so the declaration is noncompliant. By comparison, the third line declares `hhh` with an `extern` keyword creating external linkage. The fourth line declares `hhh` with internal linkage, but this declaration conflicts with the first declaration of `hhh`.

#### Correction — Consistent `static` and `extern` Use

One possible correction is to use `static` and `extern` consistently:

```
static int foo = 0;
static int foo;

extern int hhh;
extern int hhh;
```

#### Linkage Conflict Between Function Declaration and Definition

```
static int fee(void);  /* Compliant - declaration: internal linkage */
int fee(void){         /* Non-compliant */
  return 1;
}

static int ggg(int);  /* Compliant - declaration: internal linkage */
extern int ggg(int x){  /* Non-compliant */
  return 1 + x;
}
```

This example shows two internal linkage violations. Because `fee` and `ggg` have internal linkage, you must use a `static` class specifier in the definition to be compliant with MISRA.

### Check Information

**Group:** Declarations and Definitions
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

### See Also

`Check MISRA C:2012 (-misra3)`

#### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 8.9

An object should be defined at block scope if its identifier only appears in a single function

## Description

### Rule Definition

*An object should be defined at block scope if its identifier only appears in a single function.*

### Rationale

If you define an object at block scope, you or someone else is less likely to access the object inadvertently outside the block.

### Polyspace Implementation

The rule checker flags `static` objects that are accessed in one function only but declared at file scope.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Object Declared at File Scope but Used in One Function

```
static int ctr;   /* Non compliant */

int checkStatus(void);
void incrementCount(void);

void incrementCount(void) {
    ctr=0;
    while(1) {
        if(checkStatus())
            ctr++;
    }
}
```

In this example, the declaration of `ctr` is noncompliant because it is declared at file scope but used only in the function `incrementCount`. Declare `ctr` in the body of `incrementCount` to be MISRA C-compliant.

## Check Information

**Group:** Declarations and Definitions
**Category:** Advisory
**AGC Category:** Advisory

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 8.10

An inline function shall be declared with the static storage class

## Description

### Rule Definition

*An inline function shall be declared with the static storage class.*

### Rationale

If you call an inline function that is declared with external linkage but not defined in the same translation unit, the function might not be inlined. You might not see the reduction in execution time that you expect from inlining.

If you want to make an inline function available to several translation units, you can still define it with the `static` specifier. In this case, place the definition in a header file. Include the header file in all the files where you want the function inlined.

### Polyspace Implementation

The rule checker flags definitions that contain the `inline` specifier without an accompanying `static` specifier.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Inlining Functions with External Linkage

```
inline double mult(int val);
inline double mult(int val) {   /* Non compliant */
    return val * 2.0;
}

static inline double div(int val);
static inline double div(int val) {  /* Compliant */
    return val / 2.0;
}
```

In this example, the definition of `mult` is noncompliant because it is inlined without the `static` storage specifier.

## Check Information

**Group:** Declarations and Definitions
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 5.9` | `Check MISRA C:2012 (-misra3)`

### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 8.11

When an array with external linkage is declared, its size should be explicitly specified

## Description

### Rule Definition

*When an array with external linkage is declared, its size should be explicitly specified*.

### Rationale

Although it is possible to declare an array with an incomplete type and access its elements, it is safer to state the size of the array explicitly. If you provide size information for each declaration, a code reviewer can check multiple declarations for their consistency. With size information, a static analysis tool can perform array bounds analysis without analyzing more than one unit.

### Polyspace Implementation

The rule checker flags arrays declared with the `extern` specifier if the declaration does not explicitly specify the array size.

### Additional Message in Report

Size of array *`array_name`* should be explicitly stated. When an array with external linkage is declared, its size should be explicitly specified.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Array Declarations

```
#include <stdint.h>

extern int32_t array1[10];    /*  Compliant  */
extern int32_t array2[];      /*  Non-compliant  */
```

In this example, two arrays are declared `array1` and `array2`. `array1` has external linkage (the `extern` keyword) and a size of 10. `array2` also has external linkage, but no specified size. `array2` is noncompliant because for arrays with external linkage, you must explicitly specify a size.

## Check Information

**Group:** Declarations and Definitions
**Category:** Advisory
**AGC Category:** Advisory

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 8.12

Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique

## Description

### Rule Definition

*Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique*.

### Rationale

An implicitly specified enumeration constant has a value one greater than its predecessor. If the first enumeration constant is implicitly specified, then its value is 0. An explicitly specified enumeration constant has the specified value.

If implicitly and explicitly specified constants are mixed within an enumeration list, it is possible for your program to replicate values. Such replications can be unintentional and can cause unexpected behavior.

### Polyspace Implementation

The rule checker flags an enumeration if it has an implicitly specified enumeration constant with the same value as another enumeration constant.

### Additional Message in Report

The constant *`constant1`* has same value as the constant *`constant2`*.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Replication of Value in Implicitly Specified Enum Constants

```
enum color1 {red_1, blue_1, green_1};   /* Compliant */
enum color2 {red_2 = 1, blue_2 = 2, green_2 = 3};       /* Compliant */
enum color3 {red_3 = 1, blue_3, green_3};     /* Compliant */
enum color4 {red_4, blue_4, green_4 = 1};     /* Non Compliant */
enum color5 {red_5 = 2, blue_5, green_5 = 2};     /* Compliant */
enum color6 {red_6 = 2, blue_6, green_6 = 2, yellow_6};     /* Non Compliant */
```

Compliant situations:

- `color1`: All constants are implicitly specified.
- `color2`: All constants are explicitly specified.
- `color3`: Though there is a mix of implicit and explicit specification, all constants have unique values.

- `color5`: The implicitly specified constants have unique values.

Noncompliant situations:

- `color4`: The implicitly specified constant `blue_4` has the same value as `green_4`.
- `color6`: The implicitly specified constant `blue_6` has the same value as `yellow_6`.

## Check Information

**Group:** Declarations and Definitions
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Avoid Violations of MISRA C:2012 Rules 8.x”
“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 8.13

A pointer should point to a const-qualified type whenever possible

## Description

### Rule Definition

*A pointer should point to a const-qualified type whenever possible*.

### Rationale

This rule ensures that you do not inadvertently use pointers to modify objects.

### Polyspace Implementation

The rule checker flags a pointer to a non-`const` function parameter if the pointer does not modify the addressed object. The assumption is that the pointer is not meant to modify the object and so must point to a `const`-qualified type. Polyspace does not raise a flag if the data pointed to by a non`const` pointer is modified by using a copy of the pointer.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Pointer That Should Point to `const`-Qualified Types

```
#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(uint16_t *p) {      /* Non-compliant */
    return *p;
}

char last_char(char * const s){     /* Non-compliant */
    return s[strlen(s) - 1u];
}

uint16_t first(uint16_t a[5]){      /* Non-compliant */
    return a[0];
}
```

This example shows three different noncompliant pointer parameters.

- In the `ptr_ex` function, `p` does not modify an object. However, the type to which `p` points is not `const`-qualified, so it is noncompliant.
- In `last_char`, the pointer `s` is `const`-qualified but the type it points to is not. This parameter is noncompliant because `s` does not modify an object.

- The function `first` does not modify the elements of the array `a`. However, the element type is not `const`-qualified, so `a` is also noncompliant.

#### Correction — Use `const` Keywords

One possible correction is to add `const` qualifiers to the definitions.

```
#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(const uint16_t *p){     /* Compliant */
    return *p;
}

char last_char(const char * const s){   /* Compliant */
    return s[strlen( s ) - 1u];
}

uint16_t first(const uint16_t a[5]) {   /* Compliant */
    return a[0];
}
```

## Check Information

**Group:** Declarations and Definitions
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

#### Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 8.14

The restrict type qualifier shall not be used

## Description

### Rule Definition

*The restrict type qualifier shall not be used*.

### Rationale

When you use a `restrict` qualifier carefully, it improves the efficiency of code generated by a compiler. It can also improve static analysis. However, when using the `restrict` qualifier, it is difficult to make sure that the memory areas operated on by two or more pointers do not overlap.

### Polyspace Implementation

The rule checker flags all uses of the `restrict` qualifier.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `restrict` Qualifier

```
void f(int n, int * restrict p, int * restrict q)/*Noncompliant*/
{
}
```

In this example, both uses of the `restrict` qualifier are flagged.

## Check Information

**Group:** Declarations and Definitions
**Category:** Required
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 9.1

The value of an object with automatic storage duration shall not be read before it has been set

## Description

**Message in Report:**

### Rule Definition

*The value of an object with automatic storage duration shall not be read before it has been set.*

### Rationale

A variable with an automatic storage duration is allocated memory at the beginning of an enclosing code block and deallocated at the end. All non-global variables have this storage duration, except those declared `static` or `extern`.

Variables with automatic storage duration are not automatically initialized and have indeterminate values. Therefore, you must not read such a variable before you have set its value through a write operation.

### Polyspace Implementation

Polyspace reports a violation of this rule if your code contains these issues:

- `Non-initialized variable`
- `Non-initialized pointer`

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Variable Read Before Initialization

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }
    return val; //Noncompliant

}
```

### Pointer Dereferenced Before Initialization

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j; //Noncompliant

    return pi;
}
```

## Check Information

**Group:** Initialization
**Category:** Mandatory
**AGC Category:** Mandatory

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 15.1` | `MISRA C:2012 Rule 15.3` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 9.2

The initializer for an aggregate or union shall be enclosed in braces

## Description

### Rule Definition

*The initializer for an aggregate or union shall be enclosed in braces.*

### Rationale

The rule applies to both objects and subobjects. For example, when initializing a structure that contains an array, the values assigned to the structure must be enclosed in braces. Within these braces, the values assigned to the array must be enclosed in another pair of braces.

Enclosing initializers in braces improves clarity of code that contains complex data structures such as multidimensional arrays and arrays of structures.

**Tip** To avoid nested braces for subobjects, use the syntax `{0}`, which sets all values to zero.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Initialization of Two-dimensional Arrays

```
void initialize(void) {
    int x[4][2] = {{0,0},{1,0},{0,1},{1,1}}; /* Compliant */
    int y[4][2] = {{0},{1,0},{0,1},{1,1}};   /* Compliant */
    int z[4][2] = {0};                       /* Compliant */
    int w[4][2] = {0,0,1,0,0,1,1,1};         /* Non-compliant */
}
```

In this example, the rule is not violated when:

- Initializers for each row of the array are enclosed in braces.
- The syntax `{0}` initializes all elements to zero.

The rule is violated when a separate pair of braces is not used to enclose the initializers for each row.

## Check Information

**Group:** Initialization
**Category:** Required
**AGC Category:** Readability

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 9.3

Arrays shall not be partially initialized

## Description

### Rule Definition

*Arrays shall not be partially initialized*.

### Rationale

Providing an explicit initialization for each array element makes it clear that every element has been considered.

### Polyspace Implementation

The checker reports a violation of this rule if an array is partially initialized at declaration. The checker allows initialization of all values using the shorthand notation `{0}`, for instance:

```
float dat2[3*3] = {0};
```

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Partial and Complete Initializations

```
void func(void) {
    int x[3] = {0,1,2};              /* Compliant */
    int y[3] = {0,1};                /* Non-compliant */
    int z[3] = {0};                  /* Compliant - exception */
    int a[30] = {[1] = 1,[15]=1};    /* Compliant - exception */
    int b[30] = {[1] = 1, 1};        /* Non-compliant */
    char c[20] = "Hello World";      /* Compliant - exception */
}
```

In this example, the rule is not violated when each array element is explicitly initialized.

The rule is violated when some elements of the array are implicitly initialized. Exceptions include the following:

- The initializer has the form `{0}`, which initializes all elements to zero.
- The array initializer consists *only* of designated initializers. Typically, you use this approach for sparse initialization.
- The array is initialized using a string literal.

## Check Information

**Group:** Initialization
**Category:** Required
**AGC Category:** Readability

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 9.4

An element of an object shall not be initialized more than once

## Description

### Rule Definition

*An element of an object shall not be initialized more than once*.

### Rationale

Designated initializers allow explicitly initializing elements of objects such as arrays in any order. However, using designated initializers, one can inadvertently initialize the same element twice and therefore overwrite the first initialization.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Array Initialization Using Designated Initializers

```
void func(void) {
    int a[5] = {-2,-1,0,1,2};               /* Compliant */
    int b[5] = {[0]=-2, [1]=-1, [2]=0, [3]=1, [4]=2}; /* Compliant */
    int c[5] = {[0]=-2, [1]=-1, [1]=0, [3]=1, [4]=2}; /* Non-compliant */

}
```

In this example, the rule is violated when the array element `c[1]` is initialized twice using a designated initializer.

### Structure Initialization Using Designated Initializers

```
struct myStruct {
    int a;
    int b;
    int c;
    int d;
};

void func(void) {
    struct myStruct struct1 = {-4,-2,2,4}; /* Compliant */
    struct myStruct struct2 = {.a=-4, .b=-2, .c=2, .d=4}; /* Compliant */
    struct myStruct struct3 = {.a=-4, .b=-2, .b=2, .d=4}; /* Non-compliant */
}
```

In this example, the rule is violated when `struct3.b` is initialized twice using a designated initializer.

## Check Information

**Group:** Initialization
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 9.5

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly

## Description

### Rule Definition

*Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.*

### Rationale

If the size of an array is not specified explicitly, it is determined by the highest index of the elements that are initialized. When using long designated initializers, it might not be immediately apparent which element has the highest index.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Using Designated Initializers Without Specifying Array Size

```
int a[5] = {[0]= 1, [2] = 1, [4]= 1, [1] = 1};        /* Compliant */
int b[] = {[0]= 1, [2] = 1, [4]= 1, [1] = 1};         /* Non-compliant */
int c[] = {[0]= 1, [1] = 1, [2]= 1, [3]=0, [4] = 1};  /* Non-compliant */

void display(int);

void main() {
    func(a,5);
    func(b,5);
    func(c,5);
}

void func(int* arr, int size) {
    for(int i=0; i<size; i++)
        display(arr[i]);
}
```

In this example, the rule is violated when the arrays b and c are initialized using designated initializers but the array size is not specified.

## Check Information

**Group:** Initialization
**Category:** Required
**AGC Category:** Readability

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 10.1

Operands shall not be of an inappropriate essential type

## Description

### Rule Definition

*Operands shall not be of an inappropriate essential type.*

### Rationale

#### What Are Essential Types?

An essential type category defines the essential type of an object or expression.

| Essential type category | Standard types |
|---|---|
| Essentially Boolean | `bool` or `_Bool` (defined in `stdbool.h`)<br>You can also define types that are essentially Boolean using the option `Effective boolean types (-boolean-types)`. |
| Essentially character | `char` |
| Essentially enum | named `enum` |
| Essentially signed | signed `char`, signed `short`, signed `int`, signed `long`, signed `long long` |
| Essentially unsigned | unsigned `char`, unsigned `short`, unsigned `int`, unsigned `long`, unsigned `long long` |
| Essentially floating | `float`, `double`, `long double` |

#### Amplification and Rationale

For operands of some operators, you cannot use certain essential types. In the table below, each row represents an operator/operand combination. If the essential type column is not empty for that row, there is a MISRA restriction when using that type as the operand. The number in the table corresponds to the rationale list after the table.

| Operation | | Essential type category of arithmetic operand | | | | | |
|---|---|---|---|---|---|---|---|
| **Operator** | **Operand** | **Boolean** | **character** | **enum** | **signed** | **unsigned** | **floating** |
| `[ ]` | integer | 3 | 4 | | | | 1 |
| `+` (unary) | | 3 | 4 | 5 | | | |
| `-` (unary) | | 3 | 4 | 5 | | 8 | |
| `+ -` | either | 3 | | 5 | | | |
| `* /` | either | 3 | 4 | 5 | | | |
| `%` | either | 3 | 4 | 5 | | | 1 |
| `< > <= >=` | either | 3 | | | | | |

| Operation | | Essential type category of arithmetic operand | | | | | |
|---|---|---|---|---|---|---|---|
| `== !=` | either | | | | | | |
| `! && \|\|` | any | | 2 | 2 | 2 | 2 | 2 |
| `<< >>` | left | 3 | 4 | 5,6 | 6 | | 1 |
| `<< >>` | right | 3 | 4 | 7 | 7 | | 1 |
| `~ & \| ^` | any | 3 | 4 | 5,6 | 6 | | 1 |
| `?:` | 1st | | 2 | 2 | 2 | 2 | 2 |
| `?:` | 2nd and 3rd | | | | | | |

**1** An expression of essentially floating type for these operands is a constraint violation.

**2** When an operand is interpreted as a Boolean value, use an expression of essentially Boolean type.

**3** When an operand is interpreted as a numeric value, do not use an operand of essentially Boolean type.

**4** When an operand is interpreted as a numeric value, do not use an operand of essentially character type. The numeric values of character data are implementation-defined.

**5** In an arithmetic operation, do not use an operand of essentially enum type. An enum object uses an implementation-defined integer type. An operation involving an enum object can therefore yield a result with an unexpected type.

**6** Perform only shift and bitwise operations on operands of essentially unsigned type. When you use shift and bitwise operations on essentially signed types, the resulting numeric value is implementation-defined.

**7** To avoid undefined behavior on negative shifts, use an essentially unsigned right-hand operand.

**8** For the unary minus operator, do not use an operand of essentially unsigned type. The implemented size of int determines the signedness of the result.

Note that for a bit-field type, if the bit-field is implemented as:

- A Boolean, the bit-field is essentially Boolean.
- Signed or unsigned type, the bit-field is essentially signed or unsigned respectively.

  The type of the bit-field is the smallest type that can represent the bit-field. For instance, the type `stmp` here is essentially 8 bits integer:

  ```
  typedef signed int mybitfield;
  typedef struct { mybitfield f1 : 1; } stmp;
  ```

### Additional Message in Report

The *`operand_name`* operand of the *`operator_name`* operator is of an inappropriate essential type category *`category_name`*.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Violation of Rule 10.1, Rationale 2: Inappropriate Operand Types for Operators That Take Essentially Boolean Operands

```
#include<stdbool.h>
extern float f32a;
extern char cha;
extern signed char s8a;
extern unsigned char u8a,u8b,ru8a;
enum enuma { a1, a2, a3 } ena, enb;
extern bool bla, blb, rbla;
void foo(void) {

    rbla = cha && bla;        /* Non-compliant: cha is essentially char  */
    enb = ena ? a1 : a2;      /* Non-compliant: ena is essentially enum  */
    rbla = s8a && bla;        /* Non-compliant: s8a is essentially signed char  */
    ena = u8a ? a1 : a2;      /* Non-compliant: u8a is essentially unsigned char  */
    rbla = f32a && bla;       /* Non-compliant: f32a is essentially float */
    rbla = bla && blb;        /* Compliant */
    ru8a = bla ? u8a : u8b;   /* Compliant */
}
```

In the noncompliant examples, rule 10.1 is violated because:

- The operator && expects only essentially Boolean operands. However, at least one of the operands used has a different type.
- The first operand of `?:` is expected to be essentially Boolean. However, a different operand type is used.

**Note** For Polyspace to detect the rule violation, you must define the type name `boolean` as an effective Boolean type. For more information, see `Effective boolean types (-boolean-types)`.

### Violation of Rule 10.1, Rationale 3: Inappropriate Boolean Operands

```
#include<stdbool.h>
enum enuma { a1, a2, a3 } ena;
enum { K1 = 1, K2 = 2 };    /* Essentially signed */
extern char cha, chb;
extern bool bla, blb, rbla;
extern signed char rs8a, s8a;
extern unsigned char u8a;

void foo(void) {

  rbla = bla * blb;      /* Non-compliant - Boolean used as a numeric value */
  rbla = bla > blb;      /* Non-compliant - Boolean used as a numeric value */

  rbla = bla && blb;     /* Compliant */
  rbla = cha > chb;      /* Compliant */
  rbla = ena > a1;       /* Compliant */
  rbla = u8a > 0U;       /* Compliant */
  rs8a = K1 * s8a;       /* Compliant - K1 obtained from anonymous enum */

}
```

In the noncompliant examples, rule 10.1 is violated because the operators * and > do not expect essentially Boolean operands. However, the operands used here are essentially Boolean.

**Note** For Polyspace to detect the rule violation, you must define the type name `boolean` as an effective Boolean type. For more information, see `Effective boolean types (-boolean-types)`.

### Violation of Rule 10.1, Rationale 4: Inappropriate Character Operands

```
extern char rcha, cha, chb;
extern unsigned char ru8a, u8a;

void foo(void) {

  rcha = cha & chb;      /* Non-compliant - char type used as a numeric value */
  rcha = cha << 1;       /* Non-compliant - char type used as a numeric value */

  ru8a = u8a & 2U;       /* Compliant */
  ru8a = u8a << 2U;      /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because the operators & and << do not expect essentially character operands. However, at least one of the operands used here has essentially character type.

### Violation of Rule 10.1, Rationale 5: Inappropriate Enum Operands

```
typedef unsigned char boolean;

enum enuma { a1, a2, a3 } rena, ena, enb;

void foo(void) {

  ena--;             /* Non-Compliant - arithmetic operation with enum type*/
  rena = ena * a1;   /* Non-Compliant - arithmetic operation with enum type*/
  ena += a1;         /* Non-Compliant - arithmetic operation with enum type*/

}
```

In the noncompliant examples, rule 10.1 is violated because the arithmetic operators --, * and += do not expect essentially enum operands. However, at least one of the operands used here has essentially enum type.

### Violation of Rule 10.1, Rationale 6: Inappropriate Signed Operand for Bitwise Operations

```
extern signed char s8a;
extern unsigned char ru8a, u8a;

void foo(void) {

  ru8a = s8a & 2;       /* Non-compliant - bitwise operation on signed type */
  ru8a = 2 << 3U;       /* Non-compliant - shift operation on signed type */

  ru8a = u8a << 2U;     /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because the & and << operations must not be performed on essentially signed operands. However, the operands used here are signed.

### Violation of Rule 10.1, Rationale 7: Inappropriate Signed Right Operand for Shift Operations

```
extern signed char s8a;
extern unsigned char ru8a, u8a;

void foo(void) {

  ru8a = u8a << s8a;    /* Non-compliant - shift magnitude uses signed type */
  ru8a = u8a << -1;     /* Non-compliant - shift magnitude uses signed type */

  ru8a = u8a << 2U;     /* Compliant */
  ru8a = u8a << 1;      /* Compliant - exception */

}
```

In the noncompliant examples, rule 10.1 is violated because the operation << does not expect an essentially signed right operand. However, the right operands used here are signed.

## Check Information

**Group:** The Essential Type Model
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 10.2` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"
"Essential Types in MISRA C:2012 Rules 10.x"

# MISRA C:2012 Rule 10.2

Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations

## Description

### Rule Definition

*Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations*.

### Rationale

Essentially character type expressions are `char` variables. Do not use `char` in arithmetic operations because the data does not represent numeric values.

It is appropriate to use `char` with addition and subtraction operations only in the following cases:

- When one operand of the addition (+) operation is a `char` and the other is a signed or unsigned `char`, `short`, `int`, `long` or `long long`. In this case, the operation returns a `char`.
- When the first operand of the subtraction (-) operation is a `char` and the second is a signed or unsigned `char`, `short`, `int`, `long` or `long long`. If both operands are `char`, the operation returns a *standard* type. Otherwise, the operation returns a `char`.

The above uses allow manipulation of character data such as conversion between lowercase and uppercase characters or conversion between digits and their ordinal values.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

### Additional Message in Report

- The *`operand_name`* operand of the `+` operator applied to an expression of essentially character type shall have essentially signed or unsigned type.
- The right operand of the `-` operator applied to an expression of essentially character type shall have essentially signed or unsigned or character type.
- The left operand of the `-` operator shall have essentially character type if the right operand has essentially character type.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Inappropriate use of `char` with Addition and Subtraction Operators

```
#include<stdint.h>
typedef double float64_t;
extern uint8_t u8a;
extern int8_t s8a;
extern int16_t s16a;
extern int32_t s32a;
extern float64_t fla;

void foo ( void )
{
    char cha;

    s16a = s16a - 'a';  /* Noncompliant*/

    cha = '0' + fla;    /* Noncompliant*/

    cha = cha  + ':';   /* Noncompliant*/
}
```

- You cannot subtract a `char`-type variable from an integer. When you subtract `'a'` from the integer `s16a`, Polyspace raises a violation.
- In addition operations, `char` type variables can only be added to integer type variables. When you add the floating point number `fla` to `'0'`, Polyspace raises a violation.
- The arithmetic operation `cha+':'` is not a conversion from upper to lower case or from digit to cardinal value. Polyspace raises a violation when `char` variables are used in arithmetic expressions.

### Permissible use of `char` in Arithmetic Operation

```
#include<stdint.h>
typedef double float64_t;
extern uint8_t u8a;
extern int8_t s8a;
extern int16_t s16a;
extern int32_t s32a;
void foo ( void )
{
    char cha;

    cha = '0' + u8a;     /* Compliant*/

    cha = s8a + '0';     /* Compliant*/

    s32a = cha  - '0';   /* Compliant*/

    cha = '0' - s8a;     /* Compliant*/

    cha++;               /* Compliant*/
}
```

`char` type variables can be used in certain addition or subtraction operations to perform `char` data manipulations. For instance:

- You can add an unsigned integer `u8a` to the `char` type data `'0'` to convert from `'0'` to a different character.
- Similarly, you can add the signed integer `s8a` to `'0'` to perform a desired character conversion.
- You can also subtract `s8a` from the `char` data `'0'`.
- Incrementing and decrementing `char` data is also permissible.

## Check Information

**Group:** The Essential Type Model
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 10.1` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"
"Essential Types in MISRA C:2012 Rules 10.x"

# MISRA C:2012 Rule 10.3

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category

## Description

### Rule Definition

*The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.*

### Rationale

The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

### Polyspace Implementation

The checker raises a violation if an expression is assigned to a variable with a narrower essential type or a different essential type category.

The checker does not raise a violation of this rule:

- If an object is assigned the constant zero corresponding to its essential type. This acceptable zero value is `0` for integral types, `0.0` for a `double`, and `'\0'` for `char`.
- When a variable of aggregate type such as an array is initialized using the shorthand notation `{0}`, for instance:

  ```
  float dat2[3*3] = {0};
  ```
- If the macros `TRUE`/`true` and `FALSE`/`false` with the corresponding boolean value is assigned to a `bool` variable. Polyspace raises a violation if these macros are spelled with mixed case.
- If a signed constant is assigned to an unsigned variable but the signed constant has the same representation as its unsigned equivalent. For instance, the checker does not flag statements such as:

  ```
  unsigned int u = 1;
  ```

Code generation tools might use the boolean values `true/false` with integer literals `1/0` interchangeably, resulting in violation of this rule. Because this rule is advisory when used in AGC mode, you might want to justify such defects. See “Annotate Code and Hide Known or Acceptable Results”.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Initializing Variables to Zero

```
#include<stdint.h>
#include <stdbool.h>
#define FALSE 0
#define TRUE 1
void init_integer(){
    int8_t a1= 0;
    int16_t a2= 0;
    int32_t a3= 0;
    uint8_t a4= 0;
    uint16_t a5= 0;
    uint32_t a6= 0;
}
void initiate(){
    float b = 0.0/*Noncompliant*/;
    double c = 0.0;
    bool flag1 = FALSE;
    bool flag2 = FALSE;
    char ch = 0 /*Noncompliant*/;
    char ch2 = '\0';
    unsigned char uch = 0;
}
```

This example shows how to initiate variables with a zero constant.

- For integral types of various sizes, initiating the variables with `0` is compliant with this rule.
- Initiating the `double` with `0.0` and the `char` with `'\0'` are also compliant with this rule.
- Because the essential type of a `char` is not integral, initiating the `char` object `ch` with `0` is not compliant with this rule.
- The essential type of an `unsigned char` is integral. Initiating the `unsigned char uch` with `0` is compliant with this rule.

## Check Information

**Group:** The Essential Type Model
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 10.4` | `MISRA C:2012 Rule 10.5` | `MISRA C:2012 Rule 10.6` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"
"Essential Types in MISRA C:2012 Rules 10.x"

# MISRA C:2012 Rule 10.4

Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category

## Description

### Rule Definition

*Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category*.

### Rationale

The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

### Polyspace Implementation

The checker raises a violation of this rule if the two operands of an operation have different essential types. The checker message states the types detected on the two sides of the operation.

The checker does not raise a violation of this rule:

- If one of the operands is the constant zero.
- If one of the operands is a signed constant and the other operand is unsigned, and the signed constant has the same representation as its unsigned equivalent.

  For instance, the statement `u8b = u8a + 3;`, where `u8a` and `u8b` are `unsigned char` variables, does not violate the rule because the constants `3` and `3U` have the same representation.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Operands with Different Essential Types

```
#define S64_MAX (9223372036854775807LL)
#define S64_MIN (-9223372036854775808LL)
long long input_s64_a, input_s64_b, result_s64;

void my_func(void){
   if (input_s64_a < S64_MIN + input_s64_b) { //Noncompliant: 2 violations
      result_s64 = S64_MIN;
   }
}
```

In this example, the type of `S64_MIN` is essentially unsigned. The value `9223372036854775808LL` is one more than the largest value that can be represented by a 64-bit variable. Therefore, the value overflows and the result wraps around to a negative value, so `-9223372036854775808LL` is essentially unsigned.

The operation `input_s64_a < S64_MIN + input_s64_b` violates the rule twice.

- The `+` operation violates the rule. The left operand is essentially unsigned and the right operand is signed.
- The `<` operation also violates the rule. As a result of type promotion, the result of the `+` operation is essentially unsigned. Now, the left operand of the `<` operation is essentially signed but the right operand is essentially unsigned.

## Check Information

**Group:** The Essential Type Model
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 10.3` | `MISRA C:2012 Rule 10.7` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"
"Essential Types in MISRA C:2012 Rules 10.x"

# MISRA C:2012 Rule 10.5

The value of an expression should not be cast to an inappropriate essential type

## Description

### Rule Definition

*The value of an expression should not be cast to an inappropriate essential type.*

### Rationale

### Converting Between Variable Types

| | | From | | | | | |
|---|---|---|---|---|---|---|---|
| | | **Boolean** | **character** | **enum** | **signed** | **unsigned** | **floating** |
| To | **Boolean** | | Avoid | Avoid | Avoid | Avoid | Avoid |
| | **character** | Avoid | | | | | Avoid |
| | **enum** | Avoid | Avoid | Avoid | Avoid | Avoid | Avoid |
| | **signed** | Avoid | | | | | |
| | **unsigned** | Avoid | | | | | |
| | **floating** | Avoid | Avoid | | | | |

Some inappropriate explicit casts are:

- In C99, the result of a cast of assignment to `_Bool` is always 0 or 1. This result is not necessarily the case when casting to another type which is defined as essentially Boolean.
- A cast to an essential enum type may result in a value that does not lie within the set of enumeration constants for that type.
- A cast from essential Boolean to any other type is unlikely to be meaningful.
- Converting between floating and character types is not meaningful as there is no precise mapping between the two representations.

Some acceptable explicit casts are:

- To change the type in which a subsequent arithmetic operation is performed.
- To truncate a value deliberately.
- To make a type conversion explicit in the interests of clarity.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** The Essential Type Model
**Category:** Advisory
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 10.3` | `MISRA C:2012 Rule 10.8` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"
"Essential Types in MISRA C:2012 Rules 10.x"

# MISRA C:2012 Rule 10.6

The value of a composite expression shall not be assigned to an object with wider essential type

## Description

### Rule Definition

*The value of a composite expression shall not be assigned to an object with wider essential type.*

### Rationale

A *composite expression* is a nonconstant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary `+`, binary `-`)
- Bitwise (&, `|`, `^`)
- Shift (<<, >>)
- Conditional (`?`, `:`)

Unary operators such as ~ and unary `+` or `-` are also considered composite operators.

If you assign the result of a composite expression to a larger type, the implicit conversion can result in loss of value, sign, precision, or layout.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** The Essential Type Model
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 10.3` | `MISRA C:2012 Rule 10.7` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”
“Essential Types in MISRA C:2012 Rules 10.x”

# MISRA C:2012 Rule 10.7

If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type

## Description

### Rule Definition

*If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed, then the other operand shall not have wider essential type.*

### Rationale

A *composite expression* is a nonconstant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary `+`, binary `-`)
- Bitwise (&, `|`, `^`)
- Shift (<<, >>)
- Conditional (`?`, `:`)

Restricting implicit conversion on composite expressions mean that sequences of arithmetic operations within expressions must use the same essential type. This restriction reduces confusion and avoids loss of value, sign, precision, or layout. However, this rule does not imply that all operands in an expression are of the same essential type.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

### Additional Message in Report

- The right operand shall not have wider essential type than the left operand which is a composite expression.
- The left operand shall not have wider essential type than the right operand which is a composite expression.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** The Essential Type Model
**Category:** Required
**AGC Category:** Advisory

## See Also

`Check MISRA C:2012 (-misra3)`

**Topics**

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”
“Essential Types in MISRA C:2012 Rules 10.x”

# MISRA C:2012 Rule 10.8

The value of a composite expression shall not be cast to a different essential type category or a wider essential type

## Description

### Rule Definition

*The value of a composite expression shall not be cast to a different essential type category or a wider essential type.*

### Rationale

A *composite expression* is a non-constant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary `+`, binary `-`)
- Bitwise (&, `|`, `^`)
- Shift (<<, >>)
- Conditional (`?`, `:`)

Unary operators such as ~ and unary `+` or `-` are also considered composite operators.

Casting to a wider type is not permitted because the result may vary between implementations. Consider this expression:

```
(uint32_t) (u16a +u16b);
```

On a 16-bit machine the addition is performed in 16 bits. The result is wrapped before it is cast to 32 bits. On a 32-bit machine, the addition takes place in 32 bits and preserves high-order bits that are lost on a 16-bit machine. Casting to a narrower type with the same essential type category is acceptable as the explicit truncation of the results always leads to the same loss of information.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

### Polyspace Implementation

The rule checker raises a defect only if the result of a composite expression is cast to a different or wider essential type. Unary operators are not considered as composite operators.

For instance, in this example, a violation is shown in the first assignment to `i` but not the second. In the first assignment, a composite expression `i+1` is directly cast from a signed to an unsigned type. In the second assignment, the composite expression is first cast to the same type and then the result is cast to a different type.

```
typedef int int32_T;
typedef unsigned char uint8_T;
...
...
int32_T i;
```

```
i = (uint8_T)(i+1); /* Noncompliant */
i = (uint8_T)((int32_T)(i+1)); /* Compliant */
```

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Casting to Different or Wider Essential Type

```
extern unsigned short ru16a, u16a, u16b;
extern unsigned int   u32a, ru32a;
extern signed int     s32a, s32b;

void foo(void)
{
  ru16a  = (unsigned short) (u32a + u32a);/* Compliant                           */
  ru16a += (unsigned short) s32a;    /* Compliant - s32a is not composite        */
  ru32a  = (unsigned int) (u16a + u16b);  /* Noncompliant - wider essential type */
}
```

In this example, rule 10.8 is violated in the following cases:

- `s32a` and `s32b` are essentially `signed` variables. However, the result `( s32a + s32b )` is cast to an essentially `unsigned` type.
- `u16a` and `u16b` are essentially `unsigned short` variables. However, the result `( s32a + s32b )` is cast to a wider essential type, `unsigned int`.

## Check Information

**Group:** The Essential Type Model
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 10.5` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 11.1

Conversions shall not be performed between a pointer to a function and any other type

## Description

### Rule Definition

*Conversions shall not be performed between a pointer to a function and any other type.*

### Rationale

The rule forbids the following two conversions:

- Conversion from a function pointer to any other type. This conversion causes undefined behavior.
- Conversion from a function pointer to another function pointer, if the function pointers have different argument and return types.

  The conversion is forbidden because calling a function through a pointer with incompatible type results in undefined behavior.

### Polyspace Implementation

Polyspace considers both explicit and implicit casts when checking this rule. However, casts from `NULL` or `(void*)0` do not violate this rule.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Cast between two function pointers

```
typedef void (*fp16) (short n);
typedef void (*fp32) (int n);

#include <stdlib.h>                     /* To obtain macro  NULL */

void func(void) {   /* Exception 1 - Can convert a null pointer
                     * constant into a pointer to a function */
  fp16 fp1 = NULL;                 /* Compliant - exception  */
  fp16 fp2 = (fp16) fp1;           /* Compliant */
  fp32 fp3 = (fp32) fp1;           /* Non-compliant */
  if (fp2 != NULL) {}              /* Compliant - exception  */
  fp16 fp4 = (fp16) 0x8000;        /* Non-compliant - integer to
                                    * function pointer */}
```

In this example, the rule is violated when:

- The pointer `fp1` of type `fp16` is cast to type `fp32`. The function pointer types `fp16` and `fp32` have different argument types.

- An integer is cast to type `fp16`.

The rule is not violated when function pointers `fp1` and `fp2` are cast to `NULL`.

### Check Information

**Group:** Pointer Type Conversions
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

### See Also

`Check MISRA C:2012 (-misra3)`

#### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 11.2

Conversions shall not be performed between a pointer to an incomplete type and any other type

## Description

### Rule Definition

*Conversions shall not be performed between a pointer to an incomplete type and any other type.*

### Rationale

An incomplete type is a type that does not contain sufficient information to determine its size. For example, the statement `struct s;` describes an incomplete type because the fields of `s` are not defined. The size of a variable of type `s` cannot be determined.

Conversions to or from a pointer to an incomplete type result in undefined behavior. Typically, a pointer to an incomplete type is used to hide the full representation of an object. This encapsulation is broken if another pointer is implicitly or explicitly cast to such a pointer.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Casts from incomplete type

```
#include <stdio.h>
struct s *sp;
struct t *tp;
short  *ip;
struct ct *ctp1;
struct ct *ctp2;


void foo(void) {

    ip = (short *) sp;            /* Non-compliant */
    sp = (struct s *) 1234;       /* Non-compliant */
    tp = (struct t *) sp;         /* Non-compliant */
    ctp1 = (struct ct *) ctp2;    /* Compliant */

    /* You can convert a null pointer constant to
     * a pointer to an incomplete type */
    sp = NULL;                    /* Compliant - exception  */

    /* A pointer to an incomplete type may be converted into void */
    struct s *f(void);
    (void) f();                   /* Compliant - exception  */

}
```

In this example, types `s`, `t` and `ct` are incomplete. The rule is violated when:

- The variable `sp` with an incomplete type is cast to a basic type.
- The variable `sp` with an incomplete type is cast to a different incomplete type `t`.

The rule is not violated when:

- The variable `ctp2` with an incomplete type is cast to the same incomplete type.
- The `NULL` pointer is cast to the variable `sp` with an incomplete type.
- The return value of `f` with incomplete type is cast to `void`.

## Check Information

**Group:** Pointer Type Conversions
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 11.5` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 11.3

A cast shall not be performed between a pointer to object type and a pointer to a different object type

## Description

### Rule Definition

*A cast shall not be performed between a pointer to object type and a pointer to a different object type.*

### Rationale

If a pointer to an object is cast into a pointer to a different object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.

Even if the conversion produces a pointer that is correctly aligned, the behavior can be undefined if the pointer is used to access an object.

Exception: You can convert a pointer to object type into a pointer to one of the following types:

- `char`
- `signed char`
- `unsigned char`

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant: Cast to Pointer Pointing to Object of Wider Type

```
signed   char *p1;
unsigned int *p2;

void foo(void){
  p2 = ( unsigned int * ) p1;     /* Non-compliant */
}
```

In this example, `p1` can point to a `signed char` object. However, `p1` is cast to a pointer that points to an object of wider type, `unsigned int`.

### Noncompliant: Cast to Pointer Pointing to Object of Narrower Type

```
extern unsigned int read_value ( void );
extern void display ( unsigned int n );

void foo ( void ){
  unsigned int u = read_value ( );
  unsigned short *hi_p = ( unsigned short * ) &u;    /* Non-compliant  */
  *hi_p = 0;
  display ( u );
}
```

In this example, u is an unsigned int variable. &u is cast to a pointer that points to an object of narrower type, unsigned short.

On a big-endian machine, the statement *hi_p = 0 attempts to clear the high bits of the memory location that &u points to. But, from the result of display(u), you might find that the high bits have not been cleared.

#### Noncompliant: Implicit Casting

```
typedef struct {
    int iNum1;
}A;

typedef struct {
    int iNum2;
}B;

void bar(A*);

void foo() {
    B wrappedNum2;
    bar(&wrappedNum2); /* Noncompliant*/

}
```

In this example, the B type struct object wrappedNum2 is implicitly cast into an A type struct object in the call to bar. Polyspace flags the implicit casting.

#### Compliant: Cast Adding a Type Qualifier

```
const short *p;
const volatile short *q;
void foo (void){
  q = ( const volatile short * ) p;  /* Compliant */
}
```

In this example, both p and q can point to short objects. The cast between them adds a volatile qualifier only and is therefore compliant.

## Check Information

**Group:** Pointer Type Conversions
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

MISRA C:2012 Rule 11.4 | MISRA C:2012 Rule 11.5 | MISRA C:2012 Rule 11.8 | Check MISRA C:2012 (-misra3)

**Topics**

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 11.4

A conversion should not be performed between a pointer to object and an integer type

## Description

### Rule Definition

*A conversion should not be performed between a pointer to object and an integer type.*

### Rationale

Conversion between integers and pointers can cause errors or undefined behavior.

- If an integer is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.
- If a pointer is cast to an integer, the resulting value can be outside the allowed range for the integer type.

### Polyspace Implementation

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Casts between pointer and integer

```
#include <stdbool.h>

typedef unsigned char      uint8_t;
typedef          char      char_t;
typedef unsigned short     uint16_t;
typedef signed   int       int32_t;

typedef _Bool bool_t;
uint8_t *PORTA = (uint8_t *) 0x0002;            /* Non-compliant */

void foo(void) {

    char_t c = 1;
    char_t *pc = &c;                              /* Compliant */


    uint16_t ui16   = 7U;
    uint16_t *pui16 = &ui16;                      /* Compliant */
    pui16 = (uint16_t *) ui16;                    /* Non-compliant */
    uint16_t *p;
    int32_t addr = (int32_t) p;                  /* Non-compliant */
    bool_t b = (bool_t) p;                       /* Non-compliant */
    enum etag { A, B } e = ( enum etag ) p;      /* Non-compliant */
}
```

In this example, the rule is violated when:

- The integer `0x0002` is cast to a pointer.

  If the integer defines an absolute address, it is more common to assign the address to a pointer in a header file. To avoid the assignment being flagged, you can then exclude headers files from coding rules checking. For more information, see `Do not generate results for (-do-not-generate-results-for)`.
- The pointer `p` is cast to integer types such as `int32_t`, `bool_t` or `enum etag`.

The rule is not violated when the address `&ui16` is assigned to a pointer.

## Check Information

**Group:** Pointer Type Conversions
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 11.3` | `MISRA C:2012 Rule 11.7` | `MISRA C:2012 Rule 11.9` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 11.5

A conversion should not be performed from pointer to void into pointer to object

## Description

### Rule Definition

*A conversion should not be performed from pointer to void into pointer to object.*

### Rationale

If a pointer to void is cast into a pointer to an object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior. However, such a cast can sometimes be necessary, for example, when using memory allocation functions.

### Polyspace Implementation

Casts or implicit conversions from NULL or (void*)0 do not generate a warning.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Cast from Pointer to void

```
void foo(void) {

    unsigned int  u32a = 0;
    unsigned int  *p32 = &u32a;
    void          *p;
    unsigned int  *p16;

    p   = p32;                /* Compliant - pointer to uint32_t
                               *            into pointer to void */
    p16 = p;                  /* Non-compliant */

    p   = (void *) p16;       /* Compliant */
    p32 = (unsigned int *) p; /* Non-compliant */
}
```

In this example, the rule is violated when the pointer p of type void* is cast to pointers to other types.

The rule is not violated when p16 and p32, which are pointers to non-void types, are cast to void*.

## Check Information

**Group:** Pointer Type Conversions
**Category:** Advisory

**AGC Category:** Advisory

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 11.2` | `MISRA C:2012 Rule 11.3` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 11.6

A cast shall not be performed between pointer to void and an arithmetic type

## Description

### Rule Definition

*A cast shall not be performed between pointer to void and an arithmetic type*.

### Rationale

Conversion between integer types and pointers to `void` can cause errors or undefined behavior.

- If an integer type is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.
- If a pointer is cast to an arithmetic type, the resulting value can be outside the allowed range for the type.

Conversion between non-integer arithmetic types and pointers to `void` is undefined.

### Polyspace Implementation

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Casts Between Pointer to void and Arithmetic Types

```
void foo(void) {

    void          *p;
    unsigned int  u;
    unsigned short r;

    p = (void *) 0x1234u;            /* Non-compliant - undefined */
    u = (unsigned int) p;            /* Non-compliant - undefined */

    p = (void *) 0;                  /* Compliant - Exception */

}
```

In this example, `p` is a pointer to `void`. The rule is violated when:

- An integer value is cast to `p`.
- `p` is cast to an `unsigned int` type.

The rule is not violated if an integer constant with value 0 is cast to a pointer to `void`.

## Check Information

**Group:** Pointer Type Conversions
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 11.7

A cast shall not be performed between pointer to object and a non-integer arithmetic type

## Description

### Rule Definition

*A cast shall not be performed between pointer to object and a non-integer arithmetic type.*

### Rationale

This rule covers types that are essentially Boolean, character, enum or floating.

- If an essentially Boolean, character or enum variable is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior. If a pointer is cast to one of those types, the resulting value can be outside the allowed range for the type.
- Casts to or from a pointer to a floating type results in undefined behavior.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Casts from Pointer to Non-Integer Arithmetic Types

```
int foo(void) {

    short *p;
    float  f;
    long  *l;

    f = (float)   p;              /* Non-compliant */
    p = (short *) f;              /* Non-compliant */

    l = (long *)  p;              /* Compliant */
}
```

In this example, the rule is violated when:

- The pointer `p` is cast to `float`.
- A `float` variable is cast to a pointer to `short`.

Casting between a pointer and a non-integerer variable might cause a compilation failure. Polyspace flags such casts.

The rule is not violated when the pointer `p` is cast to `long*`.

## Check Information

**Group:** Pointer Type Conversions
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 11.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 11.8

A cast shall not remove any const or volatile qualification from the type pointed to by a pointer

## Description

### Rule Definition

*A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.*

### Rationale

This rule forbids:

- Casts from a pointer to a `const` object to a pointer that does not point to a `const` object.
- Casts from a pointer to a `volatile` object to a pointer that does not point to a `volatile` object.

Such casts violate type qualification. For example, the `const` qualifier indicates the read-only status of an object. If a cast removes the qualifier, the object is no longer read-only.

### Polyspace Implementation

Polyspace flags both implicit and explicit conversions that violate this rule.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Casts That Remove Qualifiers

```
void foo(void) {

    /* Cast on simple type */
    unsigned short           x;
    unsigned short * const   cpi = &x;  /* const pointer */
    unsigned short * const  *pcpi;   /* pointer to const pointer */
    unsigned short **ppi;
    const unsigned short    *pci;    /* pointer to const */
    volatile unsigned short *pvi;    /* pointer to volatile  */
    unsigned short          *pi;

    pi = cpi;                        /* Compliant - no cast required */
    pi  = (unsigned short *)  pci;   /* Non-compliant */
    pi  = (unsigned short *)  pvi;   /* Non-compliant */
    ppi = (unsigned short **)pcpi;   /* Non-compliant */
}
```

In this example:

- The variables `pci` and `pcpi` have the `const` qualifier in their type. The rule is violated when the variables are cast to types that do not have the `const` qualifier.

- The variable `pvi` has a `volatile` qualifier in its type. The rule is violated when the variable is cast to a type that does not have the `volatile` qualifier.

Even though `cpi` has a `const` qualifier in its type, the rule is not violated in the statement `p=cpi;`. The assignment does not cause a type conversion because both `p` and `cpi` have type `unsigned short`.

## Check Information

**Group:** Pointer Type Conversions
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 11.3` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 11.9

The macro NULL shall be the only permitted form of integer null pointer constant

## Description

### Rule Definition

*The macro NULL shall be the only permitted form of integer null pointer constant.*

### Rationale

The following expressions allow the use of a null pointer constant:

- Assignment to a pointer
- The == or != operation, where one operand is a pointer
- The ?: operation, where one of the operands on either side of : is a pointer

Using NULL rather than 0 makes it clear that a null pointer constant was intended.

### Polyspace Implementation

The checker flags the assignment of the constant zero to pointers, equalities (or inequalities) comparing pointers with the constant zero, and other similar expressions listed in the MISRA C: 2012 documentation.

Following the updates in MISRA C: 2012 Technical Corrigendum 1, the checker allows the use of {0} to initialize aggregates containing only pointers, for instance, arrays of pointers or structures (or unions) with only a pointer field. If an aggregate contains multiple fields, the initialization is still flagged. In these cases, you should use the macro NULL for pointer fields and 0 for integer fields to distinguish between them.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Using 0 in Pointer Assignments and Comparisons

```
void main(void) {

    int *p1 = 0;               /* Non-compliant */
    int *p2 = ( void * ) 0;   /* Compliant     */

#define MY_NULL_1 0    /* Non-compliant */
#define MY_NULL_2 ( void * ) 0

    if ( p1 == MY_NULL_1 )
    { }
    if ( p2 == MY_NULL_2 )    /* Compliant     */
    { }

}
```

In this example, the rule is violated when the constant 0 is used instead of `(void*) 0` for pointer assignments and comparisons.

### Initialization of Aggregates with Pointer Members Using {0}

```
void init () {
    int *myArray[5] = {0}; //Compliant

    struct structPtr {
        int *ptr;
    } structPtr = {0}; //Compliant

    struct StructIntPtr {
        int data;
        int *ptr;
    } StructIntPtr = {0,0}; //Non-compliant
}
```

Following the updates in MISRA C: 2012 Technical Corrigendum 1, the checker allows the use of {0} to initialize aggregates containing only pointers such as:

- Arrays of pointers, for instance, `myArray`
- Structures with one pointer field only, for instance, `structPtr`

If an aggregate contains multiple fields, such as `StructIntPtr`, the initialization is still flagged. In these cases, you should use the macro NULL for pointer fields and 0 for integer fields to distinguish between them.

## Check Information

**Group:** Pointer Type Conversions
**Category:** Required
**AGC Category:** Readability

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 11.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 12.1

The precedence of operators within expressions should be made explicit

## Description

### Rule Definition

*The precedence of operators within expressions should be made explicit.*

### Rationale

The C language has a large number of operators and their precedence is not intuitive. Inexperienced programmers can easily make mistakes. Remove any ambiguity by using parentheses to explicitly define operator precedence.

The following table list the MISRA C definition of operator precedence for this rule.

| Description | Operator and Operand | Precedence |
|---|---|---|
| Primary | identifier, constant, string literal, (expression) | 16 |
| Postfix | `[] ()` (function call) `. -> ++`(post-increment) `--`(post-decrement) `() {}`(C99: compound literals) | 15 |
| Unary | `++`(pre-increment) `--`(pre-decrement) `& * + - ~ ! sizeof _Alignof defined` (preprocessor) | 14 |
| Cast | `()` | 13 |
| Multiplicative | `* / %` | 12 |
| Additive | `+ -` | 11 |
| Bitwise shift | `<< >>` | 10 |
| Relational | `<> <= >=` | 9 |
| Equality | `== !=` | 8 |
| Bitwise AND | `&` | 7 |
| Bitwise XOR | `^` | 6 |
| Bitwise OR | `\|` | 5 |
| Logical AND | `&&` | 4 |
| Logical OR | `\|\|` | 3 |
| Conditional | `?:` | 2 |
| Assignment | `= *= /= += -= <<= >>= &= ^= \|=` | 1 |
| Comma | `,` | 0 |

### Additional Message in Report

Operand of logical %s is not a primary expression. The precedence of operators within expressions should be made explicit.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Ambiguous Precedence in Multi-Operation Expressions

```
int a, b, c, d, x;

void foo(void) {
  x = sizeof a + b;                    /* Non-compliant - MISRA-12.1 */

  x = a == b ? a : a - b;              /* Non-compliant - MISRA-12.1 */

  x = a <<  b + c ;                    /* Non-compliant - MISRA-12.1 */

  if (a || b && c) { }                 /* Non-compliant - MISRA-12.1 */

  if ( (a>x) && (b>x) || (c>x) )   { }  /* Non-compliant - MISRA-12.1 */
}
```

This example shows various violations of MISRA rule 12.1. In each violation, if you do not know the order of operations, the code could execute unexpectedly.

#### Correction — Clarify With Parentheses

To comply with this MISRA rule, add parentheses around individual operations in the expressions. One possible solution is shown here.

```
int a, b, c, d, x;

void foo(void) {
  x = sizeof(a) + b;

  x = ( a == b ) ? a : ( a - b );

  x = a << ( b + c );

  if ( ( a || b ) && c) { }

  if ( ((a>x) && (b>x)) || (c>x) ) { }
}
```

### Ambiguous Precedence In Preprocessing Expressions

```
# if defined X && X + Y > Z    /* Non-compliant - MISRA-12.1 */
# endif
```

In this example, a violation of MISRA rule 12.1 is shown in preprocessing code. In this violation, if you do not know the correct order of operations, the results can be unexpected and cause problems.

#### Correction — Clarify with Parentheses

To comply with this MISRA rule, add parentheses around individual operations in the expressions. One possible solution is shown here.

```
# if defined (X) && ( (X + Y) > Z )
# endif
```

**Compliant Expressions Without Parentheses**

```
int a, b, c, x,i = 0;
struct {int a; } s, *ps, *pp[2];

void foo(void) {
    ps = &s;
    pp[i]-> a;          /* Compliant - no need to write (pp[i])->a */
    *ps++;              /* Compliant - no need to write *( p++ ) */

    x = f ( a + b, c ); /* Compliant - no need to write f ( (a+b),c) */

    x = a, b;           /* Compliant - parsed as ( x = a ), b */

    if (a && b && c ){  /* Compliant - all operators have
                    * the same precedence */
    }
}
```

In this example, the expressions shown have multiple operations. However, these expressions are compliant because operator precedence is already clear.

## Check Information

**Group:** Expressions
**Category:** Advisory
**AGC Category:** Advisory

## See Also

MISRA C:2012 Rule 12.2 | MISRA C:2012 Rule 12.3 | MISRA C:2012 Rule 12.4 | Check MISRA C:2012 (-misra3)

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 12.2

The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand

## Description

### Rule Definition

*The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.*

### Rationale

Consider this statement:

```
var = abc << num;
```

If `abc` is a 16-bit integer, then `num` must be in the range 0–15, (nonnegative and less than 16). If `num` is negative or greater than 16, then the shift behavior is undefined.

### Polyspace Implementation

Polyspace raises a violation when the right operand of a shift operator exceeds the range defined in this rule. When the right operand is a variable, the violation is raised unless all possible value of the operand remains within the range defined in this rule.

When a preprocessor directive performs a shift operation on a number literal, Polyspace assumes that the number is 64 bits wide. The valid shift range for such a number is between 0 and 63. For instance:

```
#if (1 << 64) //Noncompliant
//...
#endif
```

When bitfields are within a complex expression, Polyspace extends this check onto the bitfield field width or the width of the base type.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Shift Operations That Have Unacceptable Right Operand

```
void foo(void) {
  int i;
  unsigned int BitPack = 0U;

  for (i = 0; i < 32; i++) {
    BitPack |= (1U << ((unsigned int)i));  //Noncompliant
  }
}
```

In this example, the left operand 1U of the shift operator has an essential type `unsigned char`. Acceptable values for the right operand lies in the range from zero to seven. Because the right operand `i` ranges from zero to 31, Polyspace flags the shift operation.

## Check Information

**Group:** Expressions
**Category:** Required
**AGC Category:** Required

## See Also

`MISRA C:2012 Rule 12.1` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 12.3

The comma operator should not be used

## Description

### Rule Definition

*The comma operator should not be used*.

### Rationale

The comma operator can be detrimental to readability. You can often write the same code in another form.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Comma Usage in C Code

```
typedef signed int abc, xyz, jkl;
static void func1 ( abc, xyz, jkl );       /* Compliant - case 1 */
int foo(void)
{
    volatile int rd = 1;                   /* Compliant - case 2*/
    int var=0, foo=0, k=0, n=2, p, t[10];  /* Compliant - case 3*/
    int abc = 0, xyz = abc + 1;            /* Compliant - case 4*/
    int jkl = ( abc + xyz, abc + xyz );    /* Noncompliant - case 1*/
    var = 1, foo += var, n = 3;          /* Noncompliant - case 2*/
    var = (n = 1, foo = 2);              /* Noncompliant - case 3*/
    for ( int *ptr = &t[ 0 ],var = 0 ;
          var < n; ++var, ++ptr){}    /* Noncompliant - case 4*/
    if ((abc,xyz)<0) { return 1; }         /* Noncompliant - case 5*/
}
```

In this example, the code shows various uses of commas in C code.

#### Noncompliant Cases

| Case | Reason for noncompliance |
|---|---|
| 1 | When reading the code, it is not immediately obvious what `jkl` is initialized to. For example, you could infer that `jkl` has a value `abc+xyz`, `(abc+xyz)*(abc+xyz)`, `f((abc+xyz),(abc+xyz))`, and so on. |
| 2 | When reading the code, it is not immediately obvious whether `foo` has a value 0 or 1 after the statement. |
| 3 | When reading the code, it is not immediately obvious what value is assigned to `var`. |

| Case | Reason for noncompliance |
|---|---|
| 4 | When reading the code, it is not immediately obvious which values control the `for` loop. |
| 5 | When reading the code, it is not immediately obvious whether the `if` statement depends on `abc`, `xyz`, or both. |

#### Compliant Cases

| Case | Reason for compliance |
|---|---|
| 1 | Using commas to call functions with variables is allowed. |
| 2 | Comma operator is not used. |
| 3 & 4 | When using the comma for initialization, the variables and their values are immediately obvious. |

## Check Information

**Group:** Expressions
**Category:** Advisory
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 12.1` | `Check MISRA C:2012 (-misra3)`

#### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 12.4

Evaluation of constant expressions should not lead to unsigned integer wrap-around

## Description

### Rule Definition

*Evaluation of constant expressions should not lead to unsigned integer wrap-around*.

### Rationale

Unsigned integer expressions do not strictly overflow, but instead wraparound. Although there may be good reasons to use modulo arithmetic at run time, intentional use at compile time is less likely and might imply logic errors.

### Polyspace Implementation

Polyspace flags the constant expressions that might wraparound.

Different compilers might define compile-time constants differently. In the following code, `c+1u` is considered a constant expression by GCC compilers, but not by the standard C compiler.

```
const uint16_t c = 0xffffu;
uint16_t y = c + 1u;
```

Whether you see a violation of this rule in the preceding code might depend on your compiler.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Unsigned Integer Wraparounds in Constant Expression

```
#define DELAY        1000000000u
#define WIDTH        4000000000u

static void fixed_pulse ( void )
{
   int off_time32 = DELAY + WIDTH;    /*Noncompliant*/
   //...
}


static void f4 ( void )
{   const int c = 0xffffffffu;
    int y = c + 1u;                 /* Compliant*/

    //...
}
```

In this example, the constant expressions `DELAY + WIDTH;` might lead to wraparounds when compiled with the standard C compiler. Polyspace flags this expression. If you compile this code with other compilers such as GCC, the expression `c+1u` might show violations of this rule.

## Check Information

**Group:** Expressions
**Category:** Advisory
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 12.1` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 12.5

The `sizeof` operator shall not have an operand which is a function parameter declared as “array of type”

## Description

### Rule Definition

*The `sizeof` operator shall not have an operand which is a function parameter declared as “array of type”.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

The `sizeof` operator acting on an array normally returns the array size in bytes. For instance, in the following code, `sizeof(arr)` returns the size of `arr` in bytes.

```
int32_t arr[4];
size_t numberOfElements = sizeof (arr) / sizeof(arr[0]);
```

However, when the array is a function parameter, it degenerates to a pointer. The `sizeof` operator acting on the array returns the corresponding pointer size and not the array size.

The use of `sizeof` operator on an array that is a function parameter typically indicates an unintended programming error.

### Additional Message in Report

The `sizeof` operator shall not have an operand which is a function parameter declared as “array of type”.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Incorrect Use of `sizeof` Operator

```
#include <stdint.h>
int32_t glbA[] = { 1, 2, 3, 4, 5 };
void f (int32_t A[4])
{
     uint32_t numElements = sizeof(A) / sizeof(int32_t);  /* Non-compliant */
    uint32_t numElements_glbA = sizeof(glbA) / sizeof(glbA[0]);  /* Compliant */
}
```

In this example, the variable `numElements` always has the same value of 1, irrespective of the number of members that appear to be in the array (4 in this case), because `A` has type `int32_t *` and not `int32_t[4]`.

The variable numElements_glbA has the expected vale of 5 because the sizeof operator acts on the global array glbA.

### Check Information

**Group:** Expressions
**Category:** Mandatory
**AGC Category:** Mandatory

## Version History

**Introduced in R2017a**

### See Also

Check MISRA C:2012 (-misra3)

#### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 13.1

Initializer lists shall not contain persistent side effects

## Description

### Rule Definition

*Initializer lists shall not contain persistent side effects.*

### Rationale

C99 permits initializer lists with expressions that can be evaluated only at run-time. However, the order in which elements of the list are evaluated is not defined. If one element of the list modifies the value of a variable which is used in another element, the ambiguity in order of evaluation causes undefined values. Therefore, this rule requires that expressions occurring in an initializer list cannot modify the variables used in them.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Initializers with Persistent Side Effect

```
volatile int v;
int x;
int y;

void f(void) {
    int arr[2] = {x+y,x-y};  /* Compliant */
    int arr2[2] = {v,v};    /* Non-compliant */
    int arr3[2] = {x++,x+y};    /* Non-compliant */
}
```

In this example, the rule is not violated in the first initialization because the initializer does not modify either `x` or `y`. The rule is violated in the other initializations.

- In the second initialization, because `v` is volatile, the initializer can modify `v`. The initialization of `arr2` is different depending on which array element is initialized first.
- In the third initialization, the initializer modifies the variable `x`. The initialization of `arr3` is different depending on whether `x++` is evaluated earlier or later.

## Check Information

**Group:** Side Effects
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 13.2` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 13.2

The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

## Description

### Rule Definition

*The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders*.

### Rationale

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

### Polyspace Implementation

Polyspace raises a violation if an expression satisfies any of these conditions:

- The same variable is modified more than once in the expression or it is both read and written.
- The expression allows more than one order of evaluation.
- The expression contains a single `volatile` object that occurs multiple times.
- The expression contains more than one `volatile` object.

Because `volatile` objects can change their value at anytime, an expression containing multiple `volatile` variables or multiple instances of the same `volatile` variable might have different results depending on the order of evaluation.

### Additional Message in Report

The value of 'XX' depends on the order of evaluation. The value of volatile 'XX' depends on the order of evaluation because of multiple accesses.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         /* Compliant */
    COPY_ELEMENT (i++); /* Noncompliant  */
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

#### Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
    f ( i++, i );                 /* Non-compliant */
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

#### Multiple `volatile` Variables in Expression

```
struct {
    volatile float x;
    volatile float y;
} volData;

float xCopy;
float yCopy;
float res, res2;

void function4(void) {
    res = volData.x + volData.y;         //Noncompliant
    res = volData.x * volData.x;      //Noncompliant
    xCopy = volData.x;
    yCopy = volData.y;
    res = xCopy + yCopy;  //Compliant
}
```

In this example, the expression `volData.x + volData.y` is noncompliant because the expression involves multiple volatile objects. The expression consists of three operations: accessing the value of `volData.x`, accessing the value of `volData.y`, and the addition. The values of the volatile fields `x` and `y` in the `volData` structure might change at any time. The value of `res` might vary depending on which variable is read first. Because the C standard does not specify the order in which the variables are read, the value of `res` might depend on the hardware and software that you use. Polyspace flags one of the `volatile` objects in the expression. Similarly, Polyspace flags one of the `volatile` objects in the expression `volData.x * volData.x`.

To avoid the violation, assign the volatile variables to nonvolatile temporary variables and use these temporary variables in the expression.

## Check Information

**Group:** Side Effects
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Dir 4.9` | `MISRA C:2012 Rule 13.1` | `MISRA C:2012 Rule 13.3` | `MISRA C:2012 Rule 13.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 13.3

A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator

## Description

### Rule Definition

*A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.*

### Rationale

The rule is violated if the following happens in the same line of code:

- The increment or decrement operator acts on a variable.
- Another read or write operation is performed on the variable.

For example, the line y=x++ violates this rule. The ++ and = operator both act on x.

Although the operator precedence rules determine the order of evaluation, placing the ++ and another operator in the same line can reduce the readability of the code.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Increment Operator Used in Expression with Other Side Effects

```
int input(void);
int choice(void);
int operation(int, int);

int func() {
    int x = input(), y = input(), res;
    int ch = choice();
    if (choice == -1)
        return(x++);           /* Non-compliant */
    if (choice == 0) {
        res = x++ + y++;           /* Non-compliant */
        return(res);
    }
    else if (choice == 1) {
        x++;                   /* Compliant */
        y++;                   /* Compliant */
        return (x+y);
    }
    else {
        res = operation(x++,y);           /* Non-compliant */
        return(res);
    }
}
```

In this example, the rule is violated when the expressions containing the `++` operator have side effects other than that caused by the operator. For example, in the expression `return(x++)`, the other side-effect is the `return` operation.

## Check Information

**Group:** Side Effects
**Category:** Advisory
**AGC Category:** Readability

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 13.2` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 13.4

The result of an assignment operator should not be used

## Description

### Rule Definition

*The result of an assignment operator should not be used.*

### Rationale

The rule is violated if the following happens in the same line of code:

- The assignment operator acts on a variable.
- Another read or operation is performed on the result of the assignment.

For example, the line `a[x]=a[x=y];` violates this rule. The `[]` operator acts on the result of the assignment `x=y`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Result of Assignment Used

```
int x, y, b, c, d;
int a[10];
unsigned int bool_var, false=0, true=1;

int foo(void) {

    x = y;            /* Compliant - x is not used */

    a[x] = a[x = y];  /* Non-compliant - Value of x=y is used */

    if ( bool_var = false )/* Non-compliant - bool_var=false is used */
{}


    if ( bool_var == false ) {}   /* Compliant */

    if ( ( 0u == 0u ) || ( bool_var = true ) )/* Non-compliant */
                    /*- even though (bool_var=true) is not evaluated */
 {}


    if ( ( x = f () ) != 0 )/* Non-compliant - value of x=f() is used */
 {}
    a[b += c] = a[b];/* Non-compliant - value of b += c is used */
    b = c = d = 0; /* Non-compliant - value of d=0 and c=d=0 are used */

}
```

In this example, the rule is violated when the result of an assignment is used.

## Check Information

**Group:** Side Effects
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 13.2` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 13.5

The right hand operand of a logical && or || operator shall not contain persistent side effects

## Description

### Rule Definition

*The right hand operand of a logical && or || operator shall not contain persistent side effects*.

### Rationale

The right operand of an `||` operator is not evaluated if the left operand is true. The right operand of an && operator is not evaluated if the left operand is false. In these cases, if the right operand modifies the value of a variable, the modification does not take place. Following the operation, if you expect a modified value of the variable, the modification might not always happen.

### Polyspace Implementation

The rule checker reports situations where the right hand side of a logical `||` or && operator has persistent side effects. For instance, if the right hand side contains a function call and the function modifies a global variable, the rule checker reports a violation.

The rule checker does not report a violation if the right hand side contains a call to a pure function, that is, a function without side effects. The checker considers a function as pure if the function only performs simple operations such as:

- Reading a nonvolatile parameter or global variable.
- Writing to a local variable.

In addition to simple operations, if the function contains a call to another function, the checker attempts to determine if the callee is a pure function. If the callee is determined to be a pure function, the checker propagates this information and tags the calling function as a pure function (as long as the other operations in the calling function are simple operations).

The rule checker does not consider a function as pure if the function does one of the following:

- Writes to a global variable or the dereference of a parameter.
- Reads or writes to a volatile variable, or contains an `asm` block.

To determine if a function is pure, the checker needs to analyze the function definition. The checker looks for function definitions only within the same translation unit as the function call (a translation unit is a source file plus all headers included in the source). If a function definition is not found in the current translation unit, the checker does not report a violation of this rule.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Right Operand of Logical Operator with Persistent Side Effects

```
int check (int arg) {
    static int count;
    if(arg > 0) {
        count++;                  /* Persistent side effect */
        return 1;
    }
    else
        return 0;
}

int getSwitch(void);
int getVal(void);

void main(void) {
    int val = getVal();
    int mySwitch = getSwitch();
    int checkResult;

    if(mySwitch && check(val)) {   /* Non-compliant */
    }

    checkResult = check(val);
    if(checkResult && mySwitch) {  /* Compliant */
    }

    if(check(val) && mySwitch) {   /* Compliant */
    }
}
```

In this example, the rule is violated when the right operand of the && operation contains a function call. The function call has a persistent side effect because the static variable `count` is modified in the function body. Depending on `mySwitch`, this modification might or might not happen.

The rule is not violated when the left operand contains a function call. Alternatively, to avoid the rule violation, assign the result of the function call to a variable. Use this variable in the logical operation in place of the function call.

## Check Information

**Group:** Side Effects
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

**Topics**

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 13.6

The operand of the sizeof operator shall not contain any expression which has potential side effects

## Description

### Rule Definition

*The operand of the sizeof operator shall not contain any expression which has potential side effects*.

### Rationale

The argument of a `sizeof` operator is usually not evaluated at run time. If the argument is an expression, you might wrongly expect that the expression is evaluated.

### Polyspace Implementation

The rule is not violated if the argument is a `volatile` variable.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Expressions in `sizeof` Operator

```
#include <stddef.h>
int x;
int y[40];
struct S {
    int a;
    int b;
};
struct S myStruct;

void main() {
    size_t sizeOfType;
    sizeOfType = sizeof(x);         /* Compliant */
    sizeOfType = sizeof(y);         /* Compliant */
    sizeOfType = sizeof(myStruct);  /* Compliant */
    sizeOfType = sizeof(x++);       /* Non-compliant */
}
```

In this example, the rule is violated when the expression `x++` is used as argument of `sizeof` operator.

## Check Information

**Group:** Side Effects
**Category:** Mandatory
**AGC Category:** Mandatory

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 18.8` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 14.1

A loop counter shall not have essentially floating type

## Description

### Rule Definition

*A loop counter shall not have essentially floating type.*

### Rationale

When using a floating-point loop counter, accumulation of rounding errors can cause the actual number of iterations to be different than the number of iterations you expect.. This rounding error can happen when a loop step that is not a power of the floating-point radix is rounded to a value that can be represented by a float.

Even if a loop with a floating-point loop counter appears to behave correctly on one implementation, it can give a different number of iterations on another implementation.

### Polyspace Implementation

If the loop counter is a variable symbol, Polyspace checks that the counter is not a float.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### `for` Loop Counters

```
int main(void){
    unsigned int counter = 0u;
    int result = 0;
    float foo;

    // Float loop counters
    for(float foo = 0.0f; foo < 1.0f; foo +=0.001f){/* Non-compliant*/
        ++counter;
    }

    float fff = 0.0f;
    for(fff = 0.0f; fff <12.0f; fff += 1.0f){/* Non-compliant*/
        result++;
    }

    // Integer loop count
    for(unsigned int count = 0u; count < 1000u; ++count){/* Compliant */
        foo = (float) count * 0.001f;
    }
}
```

In this example, the three `for` loops show three different loop counters. The first and second `for` loops use float variables as loop counters, and therefore are not compliant. The third loop uses the integer `count` as the loop counter. Even though `count` is used as a float inside the loop, the variable remains an integer when acting as the loop index. Therefore, this `for` loop is compliant.

### `while` Loop Counters

This example shows two `while` loops. The loops use floating-point variables in the `while`-loop conditions:

- The first `while` loop uses the floating-point variable `foo` in the loop condition and inside the loop. When `foo` is updated in the loop, floating-point rounding errors can cause unexpected behavior. Polyspace reports a violation.
- The second `while` loop uses the floating point array `buffer` and two integers `iter1` and `inter2` in the loop condition. Polyspace identifies `iter1` and `iter2` as the loop counters. Because these loop counters are not floating point variables, Polyspace does not report a violation.

```
int main(void){
    unsigned int iter1 =0;
    int iter2;
    float foo;
    double buffer[2];
    double tmp;

    foo = 0.0f;
    while (foo < 1.0f){/* Non-compliant - foo used as a loop counter */
        foo += 0.001f;
    }

    //...
    while((iter1+1 < 2)&& (buffer[iter1]<buffer[iter2])){ //Compliant - loop counter is integer
        // swap buffer[iter1] and buffer[iter2]
        tmp = buffer[iter2];
        buffer[iter2] = buffer[iter1];
        buffer[iter1] = tmp;
        iter2 = iter1;
        iter1++;
    }

    return 1;
}
```

## Check Information

**Group:** Control Statement Expressions
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 14.2` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 14.2

A for loop shall be well-formed

## Description

### Rule Definition

*A for loop shall be well-formed.*

### Rationale

The `for` loop provides a flexible looping facility. You can perform other operations besides the loop counter initialization, termination, and increment in the control statement, and increment the loop counter anywhere inside the loop body. However, using a restricted loop format makes your code easier to review and to analyze.

### Polyspace Implementation

A for loop consists of a control statement with three clauses and a loop body. The checker raises a violation if:

- The first clause does not contain an initialization (except for when the clause is empty). The checker considers the last assigned variable of the first `for`-loop clause as the loop counter. If the first clause is empty, the checker considers the variable incremented or decremented in the third clause as the loop counter.
- The second clause does not contain a comparison operation involving the loop counter.
- The third clause contains an operation other than incrementing or decrementing the loop counter (separated by a comma from the increment or decrement).
- The loop counter has a data type that is not an integer or a pointer type.
- The loop counter is incremented inside the loop body.

Polyspace does not raise a violation when the second clause includes a binary operation that involves the loop counter.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Altering the Loop Counter Inside the Loop

```
void foo(void){

    for(short index=0; index < 5; index++){  /* Non-compliant */
        index = index + 3;       /* Altering the loop counter */
    }
}
```

In this example, the loop counter `index` changes inside the `for` loop. It is hard to determine when the loop terminates.

#### Correction — Use Another Variable to Terminate Early

One possible correction is to use an extra flag to terminate the loop early.

In this correction, the second clause of the `for` loop depends on the counter value, `index < 5`, and upon an additional flag, `!flag`. With the additional flag, the for loop definition and counter remain readable, and you can escape the loop early.

```
#define FALSE 0
#define TRUE  1

void foo(void){

    int flag = FALSE;

    for(short index=0; (index < 5) && !flag; index++){ /* Compliant */
        if((index % 4) == 0){
            flag = TRUE;        /* allows early termination of loop */
        }
    }
}
```

#### `for` Loops With Empty Clauses

```
void foo(void){
    for(short index = 0; ; index++) {}   /* Non-compliant */

    for(short index = 0; index < 10;) {} /* Non-compliant */

    short index;
    for(; index < 10;) {}     /* Non-compliant */

    for(; index < 10; index++) {} /* Compliant */

    for(;;){}
          /* Compliant - Exception all three clauses can be empty */
}
```

This example shows `for` loops definitions with a variety of missing clauses. To be compliant, initialize the first clause variable before the `for` loop (line 9). However, you cannot have a `for` loop without the second or third clause.

The one exception is a `for` loop with all three clauses empty, so as to allow for infinite loops.

## Check Information

**Group:** Control Statement Expressions
**Category:** Required
**AGC Category:** Readability

## See Also

`MISRA C:2012 Rule 14.1` | `MISRA C:2012 Rule 14.3` | `MISRA C:2012 Rule 14.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 14.3

Controlling expressions shall not be invariant

## Description

### Rule Definition

*Controlling expressions shall not be invariant*.

### Rationale

If the controlling expression, for example an `if` condition, has a constant value, the non-changing value can point to a programming error.

### Polyspace Implementation

The checker flags conditions in `if` or `while` statements or conditions that appear as the first operands of ternary operators (`?:`) if the conditions are invariant, for instance, evaluate always to true or false.

Polyspace Bug Finder flags some violations of MISRA C 14.3 through the `Dead code` and `Useless if` checkers.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Control Statement Expressions<br>
**Category:** Required<br>
**AGC Category:** Required

## See Also

`MISRA C:2012 Rule 2.1` | `MISRA C:2012 Rule 14.2` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 14.4

The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

## Description

### Rule Definition

*The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type*

### Rationale

Strong typing requires the controlling expression on an `if` statement or iteration statement to have *essentially Boolean* type.

### Polyspace Implementation

Polyspace does not flag integer constants, for example `if(2)`.

The analysis recognizes the Boolean types, `bool` or `_Bool` (defined in `stdbool.h`)

You can also define types that are essentially Boolean using the option `Effective boolean types (-boolean-types)`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Controlling Expression in `if`, `while`, and `for`

```
#include <stdbool.h>
#include <stdlib.h>

#define TRUE 1

typedef _Bool bool_t;
extern bool_t flag;

void foo(void){
    int *p = 1;
    int *q = 0;
    int i = 0;
    while(p){}           /* Non-compliant - p is a pointer */

    while(q != NULL){}   /* Compliant */

    while(TRUE){}        /* Compliant */
    while(flag){}        /* Compliant */

    if(i){}              /* Non-compliant - int32_t is not boolean */

    if(i != 0){}         /* Compliant */

    for(int i=-10; i;i++){}   /* Non-compliant - int32_t is not boolean */

    for(int i=0; i<10;i++){}  /* Compliant */
}
```

This example shows various controlling expressions in `while`, `if`, and `for` statements.

The noncompliant statements (the first `while`, `if`, and `for` examples), use a single non-Boolean variable. If you use a single variable as the controlling statement, it must be essentially Boolean (lines 17 and 19). Boolean expressions are also compliant with MISRA.

## Check Information

**Group:** Control Statement Expressions
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 14.2` | `MISRA C:2012 Rule 20.8` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 15.1

The goto statement should not be used

## Description

### Rule Definition

*The goto statement should not be used*.

### Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `goto` Statements

```
void foo(void) {
    int i = 0, result = 0;

label1:
    for ( i; i < 5; i++ ) {
        if (i > 2) goto label2;     /* Non-compliant */
    }

label2: {
        result++;
        goto label1;                /* Non-compliant */
    }
}
```

In this example, the rule is violated when `goto` statements are used.

## Check Information

**Group:** Control Flow
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 15.2` | `MISRA C:2012 Rule 15.3` | `MISRA C:2012 Rule 15.4` | `Check MISRA C:2012 (-misra3)`

**Topics**
"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 15.2

The goto statement shall jump to a label declared later in the same function

## Description

### Rule Definition

*The goto statement shall jump to a label declared later in the same function.*

### Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand. You can use a forward `goto` statement together with a backward one to implement iterations. Restricting backward `goto` statements ensures that you use only iteration statements provided by the language such as `for` or `while` to implement iterations. This restriction reduces visual complexity of the code.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Backward goto Statements

```
void foo(void) {
    int i = 0, result = 0;

label1:
    for ( i; i < 5; i++ ) {
        if (i > 2) goto label2;   /* Compliant */
    }

label2: {
        result++;
        goto label1;               /* Non-compliant */
    }
}
```

In this example, the rule is violated when a `goto` statement causes a backward jump to `label1`.

The rule is not violated when a `goto` statement causes a forward jump to `label2`.

## Check Information

**Group:** Control Flow
**Category:** Required
**AGC Category:** Advisory

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 15.1` | `MISRA C:2012 Rule 15.3` | `MISRA C:2012 Rule 15.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 15.3

Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement

## Description

### Rule Definition

*Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.*

### Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand. Restricting use of `goto` statements to jump between blocks or into nested blocks reduces visual code complexity.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### `goto` Statements Jump Inside Block

```
void f1(int a) {
    if(a <= 0) {
        goto L2;        /* Non-compliant - L2 in different block*/
    }

    goto L1;            /* Compliant - L1 in same block*/

    if(a == 0) {
        goto L1;        /* Compliant - L1 in outer block*/
    }

    goto L2;            /* Non-compliant - L2 in inner block*/

    L1: if(a > 0) {
            L2:;
    }
}
```

In this example, `goto` statements cause jumps to different labels. The rule is violated when:

- The label occurs in a block different from the block containing the `goto` statement.

  The block containing the label neither encloses nor is enclosed by the current block.
- The label occurs in a block enclosed by the block containing the `goto` statement.

The rule is not violated when:

- The label occurs in the same block as the block containing the `goto` statement..
- The label occurs in a block that encloses the block containing the `goto` statement..

### `goto` Statements in `switch` Block

```
void f2 ( int x, int z ) {
    int y = 0;

    switch(x) {
    case 0:
        if(x == y) {
            goto L1;  /* Non-compliant - switch-clauses are treated as blocks */
        }
        break;
    case 1:
        y = x;
        L1: ++x;
        break;
    default:
        break;
    }

}
```

In this example, the label for the `goto` statement appears to occur in a block that encloses the block containing the `goto` statement. However, for the purposes of this rule, the software considers that each `case` statement begins a new block. Therefore, the `goto` statement violates the rule.

## Check Information

**Group:** Control Flow
**Category:** Required
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 15.1` | `MISRA C:2012 Rule 15.2` | `MISRA C:2012 Rule 15.4` | `MISRA C:2012 Rule 16.1` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 15.4

There should be no more than one break or goto statement used to terminate any iteration statement

## Description

### Rule Definition

*There should be no more than one break or goto statement used to terminate any iteration statement.*

### Rationale

If you use one `break` or `goto` statement in your loop, you have one secondary exit point from the loop. Restricting number of exits from a loop in this way reduces visual complexity of your code.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### `break` Statements in Inner and Outer Loops

```
volatile int stop;

int func(int *arr, int size, int sat) {
    int i,j;
    int sum = 0;
    for (i=0; i< size; i++) {   /* Compliant  */
        if(sum >= sat)
            break;
        for (j=0; j< i; j++) {  /* Compliant */
            if(stop)
                break;
            sum += arr[j];
        }
    }
}
```

In this example, the rule is not violated in both the inner and outer loop because both loops have one `break` statement each.

### `break` and `goto` Statements in Loop

```
volatile int stop;

void displayStopMessage();

int func(int *arr, int size, int sat) {
    int i;
    int sum = 0;
    for (i=0; i< size; i++) {
        if(sum >= sat)
```

```
            break;
        if(stop)
            goto L1;   /* Non-compliant  */
        sum += arr[i];
    }

    L1: displayStopMessage();
}
```

In this example, the rule is violated because the `for` loop has one `break` statement and one `goto` statement.

### `goto` Statement in Inner Loop and `break` Statement in Outer Loop

```
volatile int stop;

void displayMessage();

int func(int *arr, int size, int sat) {
    int i,j;
    int sum = 0;
    for (i=0; i< size; i++) {
        if(sum >= sat)
            break;
        for (j=0; j< i; j++) { /* Compliant */
            if(stop)
                goto L1;  /* Non-compliant */
            sum += arr[i];
        }
    }

    L1: displayMessage();
}
```

In this example, the rule is not violated in the inner loop because you can exit the loop only through the one `goto` statement. However, the rule is violated in the outer loop because you can exit the loop through either the `break` statement or the `goto` statement in the inner loop.

## Check Information

**Group:** Control Flow
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 15.1` | `MISRA C:2012 Rule 15.2` | `MISRA C:2012 Rule 15.3` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"

“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 15.5

A function should have a single point of exit at the end

## Description

### Rule Definition

*A function should have a single point of exit at the end.*

### Rationale

This rule requires that a `return` statement must occur as the last statement in the function body. Otherwise, the following issues can occur:

- Code following a `return` statement can be unintentionally omitted.
- If a function that modifies some of its arguments has early `return` statements, when reading the code, it is not immediately clear which modifications actually occur.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### More Than One `return` Statement in Function

```
#define MAX ((unsigned int)2147483647)
#define NULL (void*)0

typedef unsigned int bool_t;
bool_t false = 0;
bool_t true = 1;

bool_t f1(unsigned short n, char *p) {          /* Non-compliant */
    if(n > MAX) {
        return false;
    }

    if(p == NULL) {
        return false;
    }

    return true;
}
```

In this example, the rule is violated because there are three `return` statements.

#### Correction — Use Variable to Store Return Value

One possible correction is to store the return value in a variable and return this variable just before the function ends.

```
#define MAX ((unsigned int)2147483647)
#define NULL (void*)0

typedef unsigned int bool_t;
bool_t false = 0;
bool_t true = 1;
bool_t return_value;

bool_t f2 (unsigned short n, char *p) {         /* Compliant */
    return_value = true;
    if(n > MAX) {
        return_value = false;
    }

    if(p == NULL) {
        return_value = false;
    }

    return return_value;
}
```

## Check Information

**Group:** Control Flow
**Category:** Advisory
**AGC Category:** Advisory

# Version History

**Introduced in R2014b**

## See Also

MISRA C:2012 Rule 17.4 | Check MISRA C:2012 (-misra3)

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 15.6

The body of an iteration-statement or a selection-statement shall be a compound statement

## Description

### Rule Definition

*The body of an iteration-statement or a selection-statement shall be a compound- statement.*

### Rationale

If the block of code associated with an iteration or selection statement is not contained in braces, you can make mistakes about the association. For example:

- You can wrongly associate a line of code with an iteration or selection statement because of its indentation.
- You can accidentally place a semicolon following the iteration or selection statement. Because of the semicolon, the line following the statement is no longer associated with the statement even though you intended otherwise.

This checker enforces the practice of adding braces following a selection or iteration statement even for a single line in the body. Later, when more lines are added, the developer adding them does not need to note the absence of braces and include them.

### Polyspace Implementation

The checker flags `for` loops where the first token following a `for` statement is not a left brace, for instance:

```
for (i=init_val; i > 0; i--)
   if (arr[i] < 0)
      arr[i] = 0;
```

Similar checks are performed for `if`, `else if`, `else`, `switch`, `for` and `do..while` statements.

The second line of the message on the **Result Details** pane indicates which statement is violating the rule. For instance, in the preceding example, there are two violations. The second line of the message points to the `for` loop for one violation and the `if` condition for another.

### Additional Message in Report

- The else keyword shall be followed by either a compound statement, or another if statement.
- An if (expression) construct shall be followed by a compound statement.
- The statement forming the body of a while statement shall be a compound statement.
- The statement forming the body of a do ... while statement shall be a compound statement.
- The statement forming the body of a for statement shall be a compound statement.
- The statement forming the body of a switch statement shall be a compound statement.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Iteration Block

```
int data_available = 1;
void f1(void) {
    while(data_available)                /* Non-compliant */
        process_data();

    while(data_available) {              /* Compliant */
        process_data();
    }
}
```

In this example, the second `while` block is enclosed in braces and does not violate the rule.

### Nested Selection Statements

```
#include<stdbool.h>
void f1(bool flag_1, bool flag_2) {
    if(flag_1)                           /* Non-compliant */
        if(flag_2)                       /* Non-compliant */
            action_1();
    else                                 /* Non-compliant */
            action_2();
}
```

In this example, the rule is violated because the `if` or `else` blocks are not enclosed in braces. Unless indented as above, it is easy to associate the `else` statement with the inner `if`.

#### Correction — Place Selection Statement Block in Braces

One possible correction is to enclose each block associated with an `if` or `else` statement in braces.

```
#include<stdbool.h>
void f1(bool flag_1, bool flag_2) {
    if(flag_1) {                         /* Compliant */
        if(flag_2) {                       /* Compliant */
            action_1();
        }
    }
    else {                               /* Compliant */
        action_2();
    }
}
```

### Spurious Semicolon After Iteration Statement

```
#include<stdbool.h>
void f1(bool flag_1) {
    while(flag_1);                       /* Non-compliant */
    {
        flag_1 = action_1();
    }
}
```

In this example, the rule is violated even though the `while` statement is followed by a block in braces. The semicolon following the `while` statement causes the block to dissociated from the `while` statement.

The rule helps detect such spurious semicolons.

### Check Information

**Group:** Control Flow
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

### See Also

`Check MISRA C:2012 (-misra3)`

#### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 15.7

All if … else if constructs shall be terminated with an else statement

## Description

### Rule Definition

*All if … else if constructs shall be terminated with an else statement.*

### Rationale

Unless there is a terminating `else` statement in an `if...elseif...else` construct, during code review, it is difficult to tell if you considered all possible results for the `if` condition.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Missing `else` Block

```
#include<stdbool.h>
void action_1(void);
void action_2(void);

void f1(bool flag_1, bool flag_2) {
    if(flag_1) {
        action_1();
    }
    else if(flag_2) {/* Non-compliant */
        action_2();
    }
}
```

In this example, the rule is violated because the `if ... else if` construct does not have a terminating `else` block.

#### Correction — Add `else` Block

To avoid the rule violation, add a terminating `else` block. This `else` block can, for instance, handle exceptions or be empty.

```
#include<stdbool.h>
bool ERROR = 0;
void action_1(void);
void action_2(void);

void f1(bool flag_1, bool flag_2) {
    if(flag_1) {
        action_1();
    }
    else if(flag_2) {
        action_2();
    }else{
        // Can be empty
        ERROR = 1;
    }
}
```

## Check Information

**Group:** Control Flow
**Category:** Required
**AGC Category:** Readability

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 16.5` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 16.1

All switch statements shall be well-formed

## Description

### Rule Definition

*All switch statements shall be well-formed*

### Rationale

The syntax for switch statements in C is not particularly rigorous and can allow complex, unstructured behavior. This rule and other rules impose a simple consistent structure on the `switch` statement.

### Polyspace Implementation

Following the MISRA specifications, the coding rules checker also raises a violation of rule 16.1 if a `switch` statement violates one of these rules: 16.2, 16.3, 16.4, 16.5 or 16.6.

### Additional Message in Report

All messages in report file begin with "MISRA-C switch statements syntax normative restriction."

- Initializers shall not be used in switch clauses.
- The child statement of a switch shall be a compound statement.
- All switch clauses shall appear at the same level.
- A switch clause shall only contain switch labels and switch clauses, and no other code.
- A switch statement shall only contain switch labels and switch clauses, and no other code.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Switch Statements
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 15.3` | `MISRA C:2012 Rule 16.2` | `MISRA C:2012 Rule 16.3` | `MISRA C:2012 Rule 16.4` | `MISRA C:2012 Rule 16.5` | `MISRA C:2012 Rule 16.6` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 16.2

A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

## Description

### Rule Definition

*A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement*

### Rationale

The C Standard permits placing a switch label (for instance, `case` or `default`) before any statement contained in the body of a switch statement. This flexibility can lead to unstructured code. To prevent unstructured code, make sure a switch label appears only at the outermost level of the body of a switch statement.

### Additional Message in Report

All messages in report file begin with "MISRA-C switch statements syntax normative restriction."

- Initializers shall not be used in switch clauses.
- The child statement of a switch shall be a compound statement.
- All switch clauses shall appear at the same level.
- A switch clause shall only contain switch labels and switch clauses, and no other code.
- A switch statement shall only contain switch labels and switch clauses, and no other code.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Switch Statements
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 16.1` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 16.3

An unconditional break statement shall terminate every switch-clause

## Description

### Rule Definition

*An unconditional break statement shall terminate every switch-clause*

### Rationale

A *switch-clause* is a case containing at least one statement. Two consecutive labels without an intervening statement is compliant with MISRA.

If you fail to end your switch-clauses with a break statement, then control flow “falls” into the next statement. This next statement can be another switch-clause, or the end of the switch. This behavior is sometimes intentional, but more often it is an error. If you add additional cases later, an unterminated switch-clause can cause problems.

### Polyspace Implementation

Polyspace raises a warning for each noncompliant `case` clause.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Switch Statements
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 16.1` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 16.4

Every switch statement shall have a default label

## Description

### Rule Definition

*Every switch statement shall have a default label*

### Rationale

The requirement for a `default` label is defensive programming. Even if your switch covers all possible values, there is no guarantee that the input takes one of these values. Statements following the `default` label take some appropriate action. If the `default` label requires no action, use comments to describe why there are no specific actions.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Switch Statement Without `default`

```
short func1(short xyz){

    switch(xyz){      /* Non-compliant - default label is required */
        case 0:
            ++xyz;
            break;
        case 1:
        case 2:
            break;
    }
    return xyz;
}
```

In this example, the switch statement does not include a `default` label, and is therefore noncompliant.

#### Correction — Add `default` With Error Flag

One possible correction is to use the `default` label to flag input errors. If your switch-clauses cover all expected input, then the default cases flags any input errors.

```
short func1(short xyz){
int errorflag = 0;
    switch(xyz){      /* Compliant */
        case 0:
            ++xyz;
            break;
        case 1:
```

```
        case 2:
            break;
        default:
            errorflag = 1;
            break;
    }
    if (errorflag == 1)
        return errorflag;
    else
        return xyz;
}
```

### Switch Statement for Enumerated Inputs

```
enum Colors{
    RED, GREEN, BLUE
};

enum Colors func2(enum Colors color){
    enum Colors next;

    switch(color){      /* Non-compliant - default label is required */
        case RED:
            next = GREEN;
            break;
        case GREEN:
            next = BLUE;
            break;
        case BLUE:
            next = RED;
            break;
    }
    return next;
}
```

In this example, the switch statement does not include a `default` label, and is therefore noncompliant. Even though this switch statement handles all values of the enumeration, there is no guarantee that color takes one of the those values.

#### Correction — Add `default`

To be compliant, add the `default` label to the end of your switch. You can use this case to flag unexpected inputs.

```
enum Colors{
    RED, GREEN, BLUE, ERROR
};

enum Colors func2(enum Colors color){
    enum Colors next;

    switch(color){      /* Compliant */
        case RED:
            next = GREEN;
            break;
        case GREEN:
            next = BLUE;
            break;
        case BLUE:
            next = RED;
            break;
        default:
            next = ERROR;
            break;
    }

    return next;
}
```

## Check Information

**Group:** Switch Statements
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 2.1` | `MISRA C:2012 Rule 16.1` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 16.5

A default label shall appear as either the first or the last switch label of a switch statement

## Description

### Rule Definition

*A default label shall appear as either the first or the last switch label of a switch statement.*

### Rationale

Using this rule, you can easily locate the `default` label within a `switch` statement.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Default Case in `switch` Statements

```
void foo(int var){

    switch(var){
        default:   /* Compliant - default is the first label */
        case 0:
            ++var;
            break;
        case 1:
        case 2:
            break;
    }

    switch(var){
        case 0:
            ++var;
            break;
        default:    /* Non-compliant - default is mixed with the case labels */
        case 1:
        case 2:
            break;
    }

    switch(var){
        case 0:
            ++var;
            break;
        case 1:
        case 2:
        default:     /* Compliant - default is the last label */
            break;
    }
    switch(var){
        case 0:
            ++var;
            break;
        case 1:
        case 2:
            break;
        default:      /* Compliant - default is the last label */
            var = 0;
            break;
    }
}
```

This example shows the same switch statement several times, each with `default` in a different place. As the first, third, and fourth switch statements show, `default` must be the first or last label. `default` can be part of a compound switch-clause (for instance, the third `switch` example), but it must be the last listed.

## Check Information

**Group:** Switch Statements
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 15.7` | `MISRA C:2012 Rule 16.1` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 16.6

Every switch statement shall have at least two switch-clauses

## Description

### Rule Definition

*Every switch statement shall have at least two switch-clauses*.

### Rationale

A switch statement with a single path is redundant and can indicate a programming error.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Switch Statements
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 16.1` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 16.7

A switch-expression shall not have essentially Boolean type

## Description

### Rule Definition

*A switch-expression shall not have essentially Boolean type*

### Rationale

The C Standard requires the controlling expression to a `switch` statement to have an integer type. Because C implements Boolean values with integer types, it is possible to have a Boolean expression control a `switch` statement. For controlling flow with Boolean types, an `if-else` construction is more appropriate.

### Polyspace Implementation

The analysis recognizes the Boolean types, `bool` or `_Bool` (defined in `stdbool.h`)

You can also define types that are essentially Boolean using the option `Effective boolean types (-boolean-types)`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Switch Statements<br>
**Category:** Required<br>
**AGC Category:** Advisory

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"<br>
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 17.1

The features of <stdarg.h> shall not be used

## Description

### Rule Definition

*The features of <stdarg.h> shall not be used..*

### Rationale

The rule forbids use of va_list, va_arg, va_start, va_end, and va_copy.

You can use these features in ways where the behavior is not defined in the Standard. For instance:

- You invoke va_start in a function but do not invoke the corresponding va_end before the function block ends.
- You invoke va_arg in different functions on the same variable of type va_list.
- va_arg has the syntax type va_arg (va_list ap, type).

  You invoke va_arg with a type that is incompatible with the actual type of the argument retrieved from ap.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use of va_start, va_list, va_arg, and va_end

```
#include<stdarg.h>
void f2(int n, ...) {
    int i;
    double val;
    va_list vl;                         /* Non-compliant */

    va_start(vl, n);                    /* Non-compliant */

    for(i = 0; i < n; i++)
    {
        val = va_arg(vl, double);         /* Non-compliant */
    }

    va_end(vl);                         /* Non-compliant */
}
```

In this example, the rule is violated because va_start, va_list, va_arg and va_end are used.

### Undefined Behavior of va_arg

```
#include <stdarg.h>
void h(va_list ap) {                    /* Non-compliant */
    double y;

    y = va_arg(ap, double );              /* Non-compliant */
}

void g(unsigned short n, ...) {
    unsigned int x;
    va_list ap;                         /* Non-compliant */

    va_start(ap, n);                    /* Non-compliant */
    x = va_arg(ap, unsigned int);       /* Non-compliant */

    h(ap);

    /* Undefined - ap is indeterminate because va_arg used in h () */
    x = va_arg(ap, unsigned int);       /* Non-compliant */

}

void f(void) {
    /* undefined - uint32_t:double type mismatch when g uses va_arg () */
    g(1, 2.0, 3.0);
}
```

In this example, `va_arg` is used on the same variable `ap` of type `va_list` in both functions `g` and `h`. In `g`, the second argument is `unsigned int` and in `h`, the second argument is `double`. This type mismatch causes undefined behavior.

## Check Information

**Group:** Function
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 17.2

Functions shall not call themselves, either directly or indirectly

## Description

### Rule Definition

*Functions shall not call themselves, either directly or indirectly.*

### Rationale

Variables local to a function are stored in the call stack. If a function calls itself directly or indirectly several times, the available stack space can be exceeded, causing serious failure. Unless the recursion is tightly controlled, it is difficult to determine the maximum stack space required.

### Polyspace Implementation

The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.

You can calculate the total number of recursion cycles using the code complexity metric `Number of Recursions`.

### Additional Message in Report

**Message in Report:** Function XX is called indirectly by YY.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Direct and Indirect Recursion

```
void foo1(int);
void foo2(int);

void foo1( int n ) {     /* Non-compliant - Indirect recursion foo1->foo2->foo1... */
    if (n > 0) {
        foo2(n);
        foo1(n);               /* Non-compliant - Direct recursion */
        n--;
    }
}

void foo2( int n ) { /* Non-compliant - Indirect recursion foo2->foo1->foo2... */
    foo1(n);
}
```

In this example, the rule is violated because of:

- Direct recursion `foo1` → `foo1`.
- Indirect recursion `foo1` → `foo2` → `foo1`.

- Indirect recursion `foo2` → `foo1` → `foo2`.

Note that the location of the rule violation is different for direct and indirect recursions:

- When a function calls itself directly, the rule violation is shown on the function call.
- When several functions are involved in an indirect recursion chain, for every function in the chain, a rule violation is shown on the function signature in the function body.

### Check Information

**Group:** Function
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

### See Also

`Number of Recursions` | `Number of Direct Recursions` | `Check MISRA C:2012 (-misra3)`

#### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 17.3

A function shall not be declared implicitly

## Description

### Rule Definition

*A function shall not be declared implicitly.*

### Rationale

An implicit declaration occurs when you call a function before declaring or defining it. When you declare a function explicitly before calling it, the compiler can match the argument and return types with the parameter types in the declaration. If an implicit declaration occurs, the compiler makes assumptions about the argument and return types. For instance, it assumes a return type of `int`. The assumptions might not agree with what you expect and cause undesired type conversions.

### Additional Message in Report

Function 'XX' has no complete visible prototype at call.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Function Not Declared Before Call

```
#include <math.h>

extern double power3 (double val, int exponent);
int getChoice(void);

double func() {
    double res;
    int ch = getChoice();
    if(ch == 0) {
        res = power(2.0, 10);    /* Non-compliant */
    }
    else if( ch==1) {
        res = power2(2.0, 10);   /* Non-compliant */
    }
    else {
        res = power3(2.0, 10);   /* Compliant */
        return res;
    }
}

double power2 (double val, int exponent) {
    return (pow(val, exponent));
}
```

In this example, the rule is violated when a function that is not declared is called in the code. Even if a function definition exists later in the code, the rule violation occurs.

The rule is not violated when the function is declared before it is called in the code. If the function definition exists in another file and is available only during the link phase, you can declare the function in one of the following ways:

- Declare the function with the `extern` keyword in the current file.
- Declare the function in a header file and include the header file in the current file.

## Check Information

**Group:** Function
**Category:** Mandatory
**AGC Category:** Mandatory

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 8.2` | `MISRA C:2012 Rule 8.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 17.4

All exit paths from a function with non-void return type shall have an explicit return statement with an expression

## Description

### Rule Definition

*All exit paths from a function with non-void return type shall have an explicit return statement with an expression.*

### Rationale

If a non-`void` function does not explicitly return a value but the calling function uses the return value, the behavior is undefined. To prevent this behavior:

- You must provide `return` statements with an explicit expression.
- You must ensure that during run time, at least one `return` statement executes.

### Additional Message in Report

Missing return value for non-void function 'XX'.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Missing Return Statement Along Certain Execution Paths

```
int absolute(int v) {
    if(v < 0) {
        return v;
    }
} // Non-compliant
```

In this example, the rule is violated because a `return` statement does not exist on all execution paths. If `v >= 0`, then the control returns to the calling function without an explicit return value.

### Return Statement Without Explicit Expression

```
#define SIZE 10
int table[SIZE];

unsigned short lookup(unsigned short v) {
    if((v < 0) || (v > SIZE)) {
        return; // Non-compliant
    }
    return table[v];
}
```

In this example, the rule is violated because the `return` statement in the `if` block does not have an explicit expression.

### Check Information

**Group:** Function
**Category:** Mandatory
**AGC Category:** Mandatory

## Version History

**Introduced in R2014b**

### See Also

`MISRA C:2012 Rule 15.5` | `Check MISRA C:2012 (-misra3)`

#### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 17.5

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements

## Description

### Rule Definition

*The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.*

### Rationale

If you use an array declarator for a function parameter instead of a pointer, the function interface is clearer because you can state the minimum expected array size. If you do not state a size, the expectation is that the function can handle an array of any size. In such cases, the size value is typically another parameter of the function, or the array is terminated with a sentinel value.

However, it is legal in C to specify an array size but pass an array of smaller size. This rule prevents you from passing an array of size smaller than the size you declared.

### Additional Message in Report

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

The argument type has *actual_size* elements whereas the parameter type expects *expected_size* elements.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Incorrect Array Size Passed to Function

```
void func(int arr[4]);

int main() {
    int arrSmall[3] = {1,2,3};
    int arr[4] = {1,2,3,4};
    int arrLarge[5] ={1,2,3,4,5};

    func(arrSmall);      /* Non-compliant */
    func(arr);           /* Compliant */
    func(arrLarge);      /* Compliant */

    return 0;
}
```

In this example, the rule is violated when `arrSmall`, which has size 3, is passed to `func`, which expects at least 4 elements.

## Check Information

**Group:** Functions
**Category:** Advisory
**AGC Category:** Readability

# Version History

**Introduced in R2015b**

## See Also

`MISRA C:2012 Rule 17.6` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 17.6

The declaration of an array parameter shall not contain the static keyword between the [ ]

## Description

### Rule Definition

*The declaration of an array parameter shall not contain the static keyword between the [ ].*

### Rationale

If you use the `static` keyword within `[]` for an array parameter of a function, you can inform a C99 compiler that the array contains a minimum number of elements. The compiler can use this information to generate efficient code for certain processors. However, in your function call, if you provide less than the specified minimum number, the behavior is not defined.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use of `static` Keyword Within `[]` in Array Parameter

```
extern int arr1[20];
extern int arr2[10];


unsigned int total (unsigned int n,
                    unsigned int arr[static 20]) { // Non-compliant

    unsigned int i;
    unsigned int sum = 0;

    for (i=0U; i < n; i++) {
        sum+= arr[i];
    }

    return sum;
}

void func (void) {
    int res, res2;
    res = total (10U, arr1); //Undefined behavior
    res2 = total (20U, arr2);
}
```

In this example, the rule is violated when the `static` keyword is used within `[]` in the array parameter of function `total`. Even if you call `total` with array arguments where the behavior is well-defined, the rule violation occurs.

## Check Information

**Group:** Function
**Category:** Mandatory
**AGC Category:** Mandatory

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 17.7

The value returned by a function having non-void return type shall be used

## Description

### Rule Definition

*The value returned by a function having non-void return type shall be used*.

### Rationale

You can unintentionally call a function with a non-`void` return type but not use the return value. Because the compiler allows the call, you might not catch the omission. This rule forbids calls to a non-`void` function where the return value is not used. If you do not intend to use the return value of a function, explicitly cast the return value to `void`.

### Polyspace Implementation

The checker flags functions with non-`void` return if the return value is not used or not explicitly cast to a `void` type.

The checker does not flag the functions `memcpy`, `memset`, `memmove`, `strcpy`, `strncpy`, `strcat`, `strncat` because these functions simply return a pointer to their first arguments.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Used and Unused Return Values

```
unsigned int cutOff(unsigned int val) {
    if (val > 10 && val < 100) {
        return val;
    }
    else {
        return 0;
    }
}

unsigned int getVal(void);

void func2(void) {
    unsigned int val = getVal(), res;
    cutOff(val);           /* Non-compliant */
    res = cutOff(val);     /* Compliant */
    (void)cutOff(val);     /* Compliant */
}
```

In this example, the rule is violated when the return value of `cutOff` is not used subsequently.

The rule is not violated when the return value is:

- Assigned to another variable.
- Explicitly cast to `void`.

### Check Information

**Group:** Function
**Category:** Required
**AGC Category:** Readability

## Version History

**Introduced in R2014b**

### See Also

`MISRA C:2012 Rule 2.2` | `Check MISRA C:2012 (-misra3)`

#### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 17.8

A function parameter should not be modified

## Description

### Rule Definition

*A function parameter should not be modified.*

### Rationale

When you modify a parameter, the function argument corresponding to the parameter is not modified. However, you or another programmer unfamiliar with C can expect by mistake that the argument is also modified when you modify the parameter.

### Polyspace Implementation

Polyspace reports a violation of this rule when you modify the function parameters in the body of the function. Polyspace does not report a violation of this rule when a function parameter is passed by address as `const` to another function.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Function Parameter Modified

In this example, the rule is violated when the parameter `param1` is modified. Polyspace reports a violation. Because `param3` is passed to `foo` as a `const`, it is not modified. Polyspace does not report a violation.

```
int input(void);
int foo(const int*);
void func(int param1, int* param2, int param3) {

    param1 = input();   /* Non-compliant */
    *param2 = input();  /* Compliant */
    foo(&param3); /*Compliant*/
}
```

The rule is not violated when a pointer parameter, such as `param2` is modified.

## Check Information

**Group:** Functions
**Category:** Advisory
**AGC Category:** Readability

## Version History

**Introduced in R2015b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 18.1

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

## Description

### Rule Definition

*A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.*

### Rationale

Using an invalid array subscript can lead to erroneous behavior of the program. Run-time derived array subscripts are especially troublesome because they cannot be easily checked by manual review or static analysis.

The C Standard defines the creation of a pointer to one beyond the end of the array. The rule permits the C Standard. Dereferencing a pointer to one beyond the end of an array causes undefined behavior and is noncompliant.

### Polyspace Implementation

Polyspace reports a violation of this rule if your code contains these issues:

- `Array access out of bounds`
- `Pointer access out of bounds`

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Pointers and Arrays
**Category:** Required
**AGC Category:** Required

## See Also

`MISRA C:2012 Dir 4.1` | `MISRA C:2012 Rule 18.4` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”

“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 18.2

Subtraction between pointers shall only be applied to pointers that address elements of the same array

## Description

### Rule Definition

*Subtraction between pointers shall only be applied to pointers that address elements of the same array.*

### Rationale

This rule applies to expressions of the form `pointer_expression1 - pointer_expression2`. The behavior is undefined if `pointer_expression1` and `pointer_expression2`:

- Do not point to elements of the same array,
- Or do not point to the element one beyond the end of the array.

### Polyspace Implementation

Polyspace reports a violation of this rule when you subtract pointers that are null or that point to elements in different arrays.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Subtracting Pointers

```
#include <stddef.h>
#include <stdint.h>

void f1 (int32_t *ptr)
{
    int32_t a1[10];
    int32_t a2[10];
    int32_t *p1 = &a1[ 1];
    int32_t *p2 = &a2[10];
    ptrdiff_t diff1, diff2, diff3;

    diff1 =  p1 - a1;   // Compliant
    diff2 =  p2 - a2;   // Compliant
    diff3 =  p1 - p2;   // Noncompliant
}
```

In this example, the three subtraction expressions show the difference between compliant and noncompliant pointer subtractions. The `diff1` and `diff2` subtractions are compliant because the pointers point to the same array. The diff3 subtraction is not compliant because p1 and p2 point to different arrays.

### Noncompliant Pointer Subtraction

```
#include <stddef.h>
#include <stdint.h>

void f1()
{
    int32_t var1, var2 = 20;
    int32_t* i;
    int32_t* j;

    i = &var1;
    j = &var2;

    int32_t diff4;

    diff4 = i - j;    //Noncompliant
}
```

In this example, the diff4 subtraction is not compliant because the pointers i and j are not pointers to an array.

## Check Information

**Group:** Pointers and Arrays
**Category:** Required
**AGC Category:** Required

## See Also

MISRA C:2012 Dir 4.1 | MISRA C:2012 Rule 18.4 | Check MISRA C:2012 (-misra3)

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 18.3

The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object

## Description

### Rule Definition

*The relational operators >, >=, <, and <= shall not be applied to objects of pointer type except where they point into the same object.*

### Rationale

If two pointers do not point to the same object, comparisons between the pointers produces undefined behavior.

You can address the element beyond the end of an array, but you cannot access this element.

### Polyspace Implementation

Polyspace reports a violation of this rule when you compare pointers that are null or that point to elements in different arrays. The relational operators for the comparison are >, <, >=, and <=.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Pointer and Array Comparisons

```
void f1(void){
    int arr1[10];
    int arr2[10];
    int *ptr1 = arr1;

    if(ptr1 < arr2){}    /* Non-compliant */
    if(ptr1 < arr1){}    /* Compliant */
}
```

In this example, `ptr1` is a pointer to `arr1`. To be compliant with rule 18.3, you can compare only `ptr1` with `arr1`. Therefore, the comparison between `ptr1` and `arr2` is noncompliant.

### Structure Comparisons

```
struct limits{
  int lower_bound;
  int upper_bound;
};

void func2(void){
    struct limits lim_1 = { 2, 5 };
    struct limits lim_2 = { 10, 5 };

    if(&lim_1.lower_bound <= &lim_2.upper_bound){}  /* Non-compliant *
    if(&lim_1.lower_bound <= &lim_1.upper_bound){}  /* Compliant */
}
```

This example defines two `limits` structures, `lim1` and `lim2`, and compares the elements. To be compliant with rule 18.3, you can compare only the structure elements within a structure. The first comparison compares the `lower_bound` of `lim1` and the `upper_bound` of `lim2`. This comparison is noncompliant because the `lim_1.lower_bound` and `lim_2.upper_bound` are elements of two different structures.

## Check Information

**Group:** Pointers and Arrays
**Category:** Required
**AGC Category:** Required

## See Also

`MISRA C:2012 Dir 4.1` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 18.4

The +, -, += and -= operators should not be applied to an expression of pointer type

## Description

### Rule Definition

*The +, -, += and -= operators should not be applied to an expression of pointer type.*

### Rationale

The preferred form of pointer arithmetic is using the array subscript syntax `ptr[expr]`. This syntax is clear and less prone to error than pointer manipulation. With pointer manipulation, any explicitly calculated pointer value has the potential to access unintended or invalid memory addresses. Array indexing can also access unintended or invalid memory, but it is easier to review.

To a new C programmer, the expression `ptr+1` can be mistakenly interpreted as one plus the address of `ptr`. However, the new memory address depends on the size, in bytes, of the pointer's target. This confusion can lead to unexpected behavior.

When used with caution, pointer manipulation using `++` can be more natural (for instance, sequentially accessing locations during a memory test).

### Polyspace Implementation

Polyspace flags operations on pointers, for example, `Pointer + Integer`, `Integer + Pointer`, `Pointer - Integer`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Pointers and Array Expressions

```
void fun1(void){
    unsigned char arr[10];
    unsigned char *ptr;
    unsigned char index = 0U;

    index = index + 1U;   /* Compliant - rule only applies to pointers */

    arr[index] = 0U;      /* Compliant */
    ptr = &arr[5];        /* Compliant */
    ptr = arr;
    ptr++;                /* Compliant - increment operator not + */
    *(ptr + 5) = 0U;      /* Non-compliant */
    ptr[5] = 0U;          /* Compliant */
}
```

This example shows various operations with pointers and arrays. The only operation in this example that is noncompliant is using the + operator directly with a pointer (line 12).

### Adding Array Elements Inside a `for` Loop

```
void fun2(void){
    unsigned char array_2_2[2][2] = {{1U, 2U}, {4U, 5U}};
    unsigned char i = 0U;
    unsigned char j = 0U;
    unsigned char sum = 0U;

    for(i = 0u; i < 2U; i++){
        unsigned char *row = array_2_2[ i ];

        for(j = 0u; j < 2U; j++){
            sum += row[ j ];                   /* Compliant */
        }
    }
}
```

In this example, the second `for` loop uses the array pointer `row` in an arithmetic expression. However, this usage is compliant because it uses the array index form.

### Pointers and Array Expressions

```
void fun3(unsigned char *ptr1, unsigned char ptr2[ ]){
    ptr1++;                /* Compliant */
    ptr1 = ptr1 - 5;       /* Non-compliant */
    ptr1 -= 5;             /* Non-compliant */
    ptr1[2] = 0U;          /* Compliant */

    ptr2++;                /* Compliant */
    ptr2 = ptr2 + 3;       /* Non-compliant */
    ptr2 += 3;             /* Non-compliant */
    ptr2[3] = 0U;          /* Compliant */
}
```

This example shows the offending operators used on pointers and arrays. Notice that the same types of expressions are compliant and noncompliant for both pointers and arrays.

If `ptr1` does not point to an array with at least six elements, and `ptr2` does not point to an array with at least 4 elements, this example violates rule 18.1.

## Check Information

**Group:** Pointers and Arrays
**Category:** Advisory
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 18.1` | `MISRA C:2012 Rule 18.2` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 18.5

Declarations should contain no more than two levels of pointer nesting

## Description

### Rule Definition

*Declarations should contain no more than two levels of pointer nesting*.

### Rationale

The use of more than two levels of pointer nesting can seriously impair the ability to understand the behavior of the code. Avoid this usage.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Pointer Nesting

```
typedef char *INTPTR;

void function(char ** arrPar[ ])    /* Non-compliant - 3 levels */
{
    char   **  obj2;            /* Compliant */
    char   *** obj3;            /* Non-compliant */
    INTPTR *   obj4;            /* Compliant */
    INTPTR * const * const obj5;    /* Non-compliant */
    char   ** arr[10];            /* Compliant */
    char   ** (*parr)[10];        /* Compliant */
    char   *  (**pparr)[10];        /* Compliant */
}

struct s{
    char *   s1;                /* Compliant */
    char **  s2;                /* Compliant */
    char *** s3;                /* Non-compliant */
};

struct s *   ps1;            /* Compliant */
struct s **  ps2;            /* Compliant */
struct s *** ps3;            /* Non-compliant */

char **  (  *pfunc1)(void);        /* Compliant */
char **  ( **pfunc2)(void);        /* Compliant */
char **  (***pfunc3)(void);        /* Non-compliant */
char *** ( **pfunc4)(void);        /* Non-compliant */
```

This example shows various pointer declarations and nesting levels. Any pointer with more than two levels of nesting is considered noncompliant.

## Check Information

**Group:** Pointers and Arrays
**Category:** Advisory
**AGC Category:** Readability

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 18.6

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist

## Description

### Rule Definition

*The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.*

### Rationale

The address of an object becomes indeterminate when the lifetime of that object expires. Any use of an indeterminate address results in undefined behavior.

### Polyspace Implementation

Polyspace flags a violation when assigning an address to a global variable, returning a local variable address, or returning a parameter address.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Address of Local Variables

```
char *func(void){
    char local_auto;
    return &local_auto ; /* Non-compliant
                          * &local_auto is indeterminate */
}
```

In this example, because `local_auto` is a local variable, after the function returns, the address of `local_auto` is indeterminate.

### Copying Pointer Addresses to Local Variables

```
char *sp;

void f(unsigned short u){
    g(&u);
}

void h(void){
    static unsigned short *q;

    unsigned short x =0u;
    q = &x;  /* Non-compliant -
```

```
              * &x stored in object with greater lifetime */
}
```

In this example, the function h stores the address of a local variable x in the a static variable q. The lifetime of the static variable q persists after the lifetime of the local variable x ends. Copying x into q is noncompliant with this rule and Polyspace flags the variable x.

## Check Information

**Group:** Pointers and Arrays
**Category:** Required
**AGC Category:** Required

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 18.7

Flexible array members shall not be declared

## Description

### Rule Definition

*Flexible array members shall not be declared*.

### Rationale

Flexible array members are usually used with dynamic memory allocation. Dynamic memory allocation is banned by Directive 4.12 and Rule 21.3 on page 21-294.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Pointers and Arrays<br>
**Category:** Required<br>
**AGC Category:** Required

## See Also

`MISRA C:2012 Rule 21.3` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”<br>
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 18.8

Variable-length array types shall not be used

## Description

### Rule Definition

*Variable-length array types shall not be used*.

### Rationale

When the size of an array declared in a block or function prototype is not an integer constant expression, you specify variable array types. Variable array types are typically implemented as a variable size object stored on the stack. Using variable type arrays can make it impossible to determine statistically the amount of memory for the stack requires.

If the size of a variable-length array is negative or zero, the behavior is undefined.

If a variable-length array must be compatible with another array type, then the size of the array types must be identical and positive integers. If your array does not meet these requirements, the behavior is undefined.

If you use a variable-length array type in a `sizeof`, it is uncertain if the array size is evaluated or not.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Pointers and Arrays<br>
**Category:** Required<br>
**AGC Category:** Required

## See Also

`MISRA C:2012 Rule 13.6` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 19.1

An object shall not be assigned or copied to an overlapping object

## Description

### Rule Definition

*An object shall not be assigned or copied to an overlapping object.*

### Rationale

When you assign an object to another object with overlapping memory, the behavior is undefined. The exceptions are:

- You assign an object to another object with exactly overlapping memory and compatible type.
- You copy one object to another using `memmove`.

### Additional Message in Report

- An object shall not be assigned or copied to an overlapping object.
- Destination and source of XX overlap, the behavior is undefined.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Assignment of Union Members

```
void func (void) {
    union {
        short i;
        int j;
    } a = {0}, b = {1};

    a.j = a.i;   /* Non-compliant */
    a = b;       /* Compliant */
}
```

In this example, the rule is violated when `a.i` is assigned to `a.j` because the two variables have overlapping regions of memory.

### Assignment of Array Segments

```
#include <string.h>

int arr[10];

void func(void) {
    memcpy (&arr[5], &arr[4], 2u * sizeof(arr[0]));    /* Non-compliant */
    memcpy (&arr[5], &arr[4], sizeof(arr[0]));         /* Compliant */
    memcpy (&arr[1], &arr[4], 2u * sizeof(arr[0]));    /* Compliant */
}
```

In this example, memory equal to twice `sizeof(arr[0])` is the memory space taken up by two array elements. If that memory space begins from `&a[4]` and `&a[5]`, the two memory regions overlap. The rule is violated when the `memcpy` function is used to copy the contents of these two overlapping memory regions.

#### Assignment Between Members of Same Aggregate Object

```
typedef struct
{
    char init_string[21];
    char new_string[101];
} string_aggr;


string_aggr my_string_aggr;

void copy()
{
    strncpy(&my_string_aggr.new_string[0], /* Non-compliant */
            &my_string_aggr.init_string[0], 100U);
}
```

In this example, the member `init_string` of the aggregate structure `my_string_aggr` consists of 21 characters. But, more than 21 characters of this member are copied to the adjoining member `new_string`, resulting in an overlapping copy.

### Check Information

**Group:** Overlapping Storage
**Category:** Mandatory
**AGC Category:** Mandatory

## Version History

**Introduced in R2014b**

### See Also

`MISRA C:2012 Rule 19.2` | `Check MISRA C:2012 (-misra3)`

#### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 19.2

The union keyword should not be used

## Description

### Rule Definition

*The union keyword should not be used*.

### Rationale

If you write to a union member and read the same union member, the behavior is well-defined. But if you read a different member, the behavior depends on the relative sizes of the members. For instance:

- If you read a union member with wider memory size, the value you read is unspecified.
- Otherwise, the value is implementation-dependent.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Possible Problems with union Keyword

```
unsigned int zext(unsigned int s)
{
    union                /* Non-compliant */
    {
        unsigned int ul;
        unsigned short us;
    } tmp;

    tmp.us = s;
    return tmp.ul;       /* Unspecified value */
}
```

In this example, the 16-bit `short` field `tmp.us` is written but the wider 32-bit `int` field `tmp.ul` is read. Using the `union` keyword can cause such unspecified behavior. Therefore, the rule forbids using the `union` keyword.

## Check Information

**Group:** Overlapping Storage<br>
**Category:** Advisory<br>
**AGC Category:** Advisory

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 19.1` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 20.1

#include directives should only be preceded by preprocessor directives or comments

## Description

### Rule Definition

*#include directives should only be preceded by preprocessor directives or comments.*

### Rationale

For better code readability, group all `#include` directives in a file at the top of the source file. Undefined behavior can occur if you use `#include` to include a standard header file within a declaration or definition or if you use part of the Standard Library before including the related standard header files.

### Polyspace Implementation

Polyspace flags text that precedes an `#include` directive. Polyspace ignores preprocessor directives, comments, spaces, or new line characters. Polyspace also ignores code that is hidden by using conditional compilation directives such as `#if` or `#ifdef`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Placing Code Before #include Directives

```
#if DEBUG

assert(0);

#endif

#include<stdlib> //Compliant


int x;

#include <conio> //Noncompliant
```

In this example, the first `#include` statement is preceded by an `assert` statement. Because the `assert` statement is hidden by the `#if` condition, Polyspace does not flag the `#include` statement. The second `#include` statement follows a variable declaration that is not hidden. Polyspace flags the second `#include` statement.

## Check Information

**Group:** Preprocessing Directives
**Category:** Advisory
**AGC Category:** Advisory

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 20.2

The ', " or \ characters and the /* or // character sequences shall not occur in a header file name

## Description

### Rule Definition

*The ', " or \ characters and the /* or // character sequences shall not occur in a header file name*.

### Rationale

The program's behavior is undefined if:

- You use `'`, `"`, `\`, `/*` or `//` between `< >` delimiters in a header name preprocessing token.
- You use `'`, `\`, `/*` or `//` between `"` delimiters in a header name preprocessing token.

Although `\` results in undefined behavior, many implementations accept `/` in its place.

### Polyspace Implementation

Polyspace flags the characters `'`, `"`, `\`, `/*` or `//` between < and > in `#include <filename>`.

Polyspace flags the characters `'`, `\`, `/*` or `//` between `"` and `"` in `#include "filename"`.

### Additional Message in Report

The ', "or \ characters and the /* or // character sequences shall not occur in a header file name.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Preprocessing Directives
**Category:** Required
**AGC Category:** Required

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 20.3

The #include directive shall be followed by either a <filename> or "filename" sequence

## Description

### Rule Definition

*The #include directive shall be followed by either a <filename> or "filename" sequence*.

### Rationale

This rule applies only after macro replacement.

The behavior is undefined if an `#include` directive does not use one of the following forms:

- `#include <filename>`
- `#include "filename"`

### Additional Message in Report

- ‘#include' expects "FILENAME" or <FILENAME>
- ‘#include_next' expects "FILENAME" or <FILENAME>
- ‘#include' does not expect string concatenation.
- ‘#include_next' does not expect string concatenation.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required
**AGC Category:** Required

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 20.4

A macro shall not be defined with the same name as a keyword

## Description

### Rule Definition

*A macro shall not be defined with the same name as a keyword.*

### Rationale

Using macros to change the meaning of keywords can be confusing. The behavior is undefined if you include a standard header while a macro is defined with the same name as a keyword.

### Additional Message in Report

- The macro *macro_name* shall not be redefined.
- The macro *macro_name* shall not be undefined.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Redefining int keyword

```
#include <stdlib.h>
#define int some_other_type /* Non-compliant - int keyword behavior altered */

//...
```

In this example, the #define violates Rule 20.4 because it alters the behavior of the int keyword. The inclusion of the standard header results in undefined behavior.

#### Correction — Rename keyword

One possible correction is to use a different keyword:

```
#include <stdlib.h>
#define int_mine some_other_type

//...
```

### Redefining keywords versus statements

```
#define while(E) for ( ; (E) ; )  /* Non-compliant - while redefined*/
#define unless(E) if ( !(E) )     /* Compliant*/

#define seq(S1, S2) do{ S1; S2;} while(false)  /* Compliant*/
#define compound(S) {S;}                       /* Compliant*/
//...
```

In this example, it is noncompliant to redefine the keyword `while`, but it is compliant to define a macro that expands to statements.

### Redefining keywords in different standards

```
#define inline // Non-compliant
```

In this example, redefining `inline` is compliant in C90, but not in C99 because `inline` is not a keyword in C90.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required
**AGC Category:** Required

## See Also

`MISRA C:2012 Rule 21.1 | Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 20.5

#undef should not be used

## Description

### Rule Definition

*#undef should not be used*.

### Rationale

`#undef` can make the software unclear which macros exist at a particular point within a translation unit.

### Additional Message in Report

#undef shall not be used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Preprocessing Directives
**Category:** Advisory
**AGC Category:** Readability

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 20.6

Tokens that look like a preprocessing directive shall not occur within a macro argument

## Description

### Rule Definition

*Tokens that look like a preprocessing directive shall not occur within a macro argument.*

### Rationale

An argument containing sequences of tokens that otherwise act as preprocessing directives leads to undefined behavior.

### Polyspace Implementation

Polyspace looks for the # character in a macro arguments (outside a string or character constant).

### Additional Message in Report

Macro argument shall not look like a preprocessing directive.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Macro Expansion Causing Non-Compliance

```
#define M( A ) printf ( #A )

#include <stdio.h>

void foo(void){
    M(
#ifdef SW         /* Non-compliant */
    "Message 1"
#else
    "Message 2"   /* Compliant - SW not defined */
#endif            /* Non-compliant */
    );
}
```

This example shows a macro definition and the macro usage. `#ifdef SW` and `#endif` are noncompliant because they look like a preprocessing directive. Polyspace does not flag `#else "Message 2"` because after macro expansion, Polyspace knows `SW` is not defined. The expanded macro is `printf ("\"Message 2\"");`

## Check Information

**Group:** Preprocessing Directives
**Category:** Required
**AGC Category:** Required

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 20.7

Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses

## Description

### Rule Definition

*Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.*

### Rationale

If you do not use parentheses, then it is possible that operator precedence does not give the results that you want when macro substitution occurs.

If you are not using a macro parameter as an expression, then the parentheses are not necessary because no operators are involved in the macro.

### Additional Message in Report

Expanded macro parameter *`param`* shall be enclosed in parentheses.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Macro Expressions

```
#define mac1(x, y) (x * y)
#define mac2(x, y) ((x) * (y))

void foo(void){
    int r;

    r = mac1(1 + 2, 3 + 4);       /* Non-compliant */
    r = mac1((1 + 2), (3 + 4));   /* Compliant */

    r = mac2(1 + 2, 3 + 4);       /* Compliant */
}
```

In this example, `mac1` and `mac2` are two defined macro expressions. The definition of `mac1` does not enclose the arguments in parentheses. In line 7, the macro expands to `r = (1 + 2 * 3 + 4);` This expression can be `(1 + (2 * 3) + 4)` or `(1 + 2) * (3 + 4)`. However, without parentheses, the program does not know the intended expression. Line 8 uses parentheses, so the line expands to `(1 + 2) * (3 + 4)`. This macro expression is compliant.

The definition of `mac2` does enclose the argument in parentheses. Line 10 (the same macro arguments in line 7) expands to `(1 + 2) * (3 + 4)`. This macro and macro expression are compliant.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required
**AGC Category:** Required

## See Also

`MISRA C:2012 Dir 4.9` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 20.8

The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1

## Description

### Rule Definition

*The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1*.

### Rationale

Strong typing requires that conditional inclusion preprocessing directives, `#if` or `#elif`, have a controlling expression that evaluates to a Boolean value.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Preprocessing Directives
**Category:** Required
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 14.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 20.9

All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation

## Description

### Rule Definition

*All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation.*

### Rationale

If attempt to use a macro identifier in a preprocessing directive, and you have not defined that identifier, then the preprocessor assumes that it has a value of zero. This value might not meet developer expectations.

### Additional Message in Report

`Identifier` is not defined.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Macro Identifiers

```
#if M == 0                   /* Non-compliant - Not defined */
#endif

#if defined (M)              /* Compliant - M is not evaluate */
#if M == 0                   /* Compliant - M is known to be defined */
#endif
#endif

#if defined (M) && (M == 0)  /* Compliant
                              * if M defined, M evaluated in ( M == 0 ) */
#endif
```

This example shows various uses of `M` in preprocessing directives. The second and third `#if` clauses check to see if the software defines `M` before evaluating `M`. The first `#if` clause does not check to see if `M` is defined, and because `M` is not defined, the statement is noncompliant.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required
**AGC Category:** Required

## See Also

`MISRA C:2012 Dir 4.9` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 20.10

The # and ## preprocessor operators should not be used

## Description

### Rule Definition

*The # and ## preprocessor operators should not be used*.

### Rationale

The order of evaluation associated with multiple #, multiple ##, or a mix of # and ## preprocessor operators is unspecified. In some cases, it is therefore not possible to predict the result of macro expansion.

The use of ## can result in obscured code.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Preprocessing Directives<br>
**Category:** Advisory<br>
**AGC Category:** Advisory

## See Also

`MISRA C:2012 Rule 1.3` | `MISRA C:2012 Rule 20.11` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 20.11

A macro parameter immediately following a # operator shall not immediately be followed by a ## operator

## Description

### Rule Definition

*A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.*

### Rationale

The order of evaluation associated with multiple #, multiple ##, or a mix of # and ## preprocessor operators, is unspecified. Rule 20.10 discourages the use of # and ##. The result of a # operator is a string literal. It is extremely unlikely that pasting this result to any other preprocessing token results in a valid token.

### Additional Message in Report

The ## preprocessor operator shall not follow a macro parameter following a # preprocessor operator.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of # and ##

```
#define A( x )    #x             /* Compliant */
#define B( x, y ) x ## y         /* Compliant */
#define C( x, y ) #x ## y     /* Non-compliant */
```

In this example, you can see three uses of the # and ## operators. You can use these preprocessing operators alone (line 1 and line 2), but using # then ## is noncompliant (line 3).

## Check Information

**Group:** Preprocessing Directives
**Category:** Required
**AGC Category:** Required

## See Also

`MISRA C:2012 Rule 20.10` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”

“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 20.12

A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators

## Description

### Rule Definition

*A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.*

### Rationale

The parameter to # or ## is not expanded prior to being used. The same parameter appearing elsewhere in the replacement text is expanded. If the macro parameter is itself subject to macro replacement, its use in mixed contexts within a macro replacement might not meet developer expectations.

### Additional Message in Report

Expanded macro parameter *`param1`* is also an operand of *`op`* operator.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Preprocessing Directives
**Category:** Required
**AGC Category:** Required

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 20.13

A line whose first token is # shall be a valid preprocessing directive

## Description

### Rule Definition

*A line whose first token is # shall be a valid preprocessing directive*

### Rationale

You typically use a preprocessing directive to conditionally exclude source code until a corresponding `#else`, `#elif`, or `#endif` directive is encountered. If your compiler does not detect a preprocessing directive because it is malformed or invalid, you can end up excluding more code than you intended.

If all preprocessing directives are syntactically valid, even in excluded code, this unintended code exclusion cannot happen.

### Additional Message in Report

Directive is not syntactically meaningful.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required
**AGC Category:** Required

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 20.14

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related

## Description

### Rule Definition

*All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related.*

### Rationale

When conditional compilation directives include or exclude blocks of code and are spread over multiple files, confusion arises. If you terminate an `#if` directive within the same file, you reduce the visual complexity of the code and the chances of an error.

If you terminate `#if` directives within the same file, you can use `#if` directives in included files

### Additional Message in Report

- '#else' not within a conditional.
- '#elseif' not within a conditional.
- '#endif' not within a conditional.

  Unterminated conditional directive.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required
**AGC Category:** Required

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 21.1

#define and #undef shall not be used on a reserved identifier or reserved macro name

## Description

### Rule Definition

*#define and #undef shall not be used on a reserved identifier or reserved macro name.*

### Rationale

Reserved identifiers and reserved macro names are intended for use by the implementation. Removing or changing the meaning of a reserved macro can result in undefined behavior. This rule applies to the following:

- Identifiers or macro names beginning with an underscore
- Identifiers in file scope described in the C Standard Library
- Macro names described in the C Standard Library as being defined in a standard header

The rule checker can flag different identifiers or macros depending on the version of the C standard used in the analysis. See `C standard version (-c-version)`. For instance, if you run a C99 analysis, the reserved identifiers and macros are defined in the ISO/IEC 9899:1999 standard, Section 7, "Library".

### Additional Message in Report

- The macro *`macro_name`* shall not be redefined.
- The macro *`macro_name`* shall not be undefined.
- The macro *`macro_name`* shall not be defined.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Defining or Undefining Reserved Identifiers

```
#undef __LINE__               /* Non-compliant - begins with _ */
#define _Guard_H 1            /* Non-compliant - begins with _ */
#undef _ BUILTIN_sqrt        /* Non-compliant - implementation may
                               * use _BUILTIN_sqrt for other purposes,
                               * e.g. generating a sqrt instruction */
#define defined               /* Non-compliant - reserved identifier */
#define errno my_errno        /* Non-compliant - library identifier */
#define isneg(x) ( (x) < 0 )  /* Compliant - rule doesn't include
                               * future library directions   */
```

## Check Information

**Group:** Standard Libraries
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 20.4` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 21.2

A reserved identifier or reserved macro name shall not be declared

## Description

### Rule Definition

*A reserved identifier or reserved macro name shall not be declared.*

### Rationale

The Standard allows implementations to treat reserved identifiers specially. If you reuse reserved identifiers, you can cause undefined behavior.

### Polyspace Implementation

- If you define a macro name that corresponds to a standard library macro, object, or function, rule 21.1 is violated.
- The rule considers tentative definitions as definitions.

### Additional Message in Report

Identifier 'XX' shall not be reused.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Standard Libraries
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 21.20

The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function

## Description

### Rule Definition

*The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

The preceding functions return a pointer to an object within the Standard Library. Implementation for this object can use a static buffer that can be modified by a second call to the same function. Therefore the value accessed through a pointer before a subsequent call to the same function can change unexpectedly.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use of Return Value from `getenv` After Another Call to `getenv`

```
#include <stdio.h>
#include <locale.h>
#include <string.h>

void f1(void)
{
    const char* res1;
    const char* res2;
    char copy[ 128 ];
    res1 = setlocale(LC_ALL, 0);
    (void) strcpy(copy, res1);
    res2 = setlocale(LC_MONETARY, "French");
    printf("%s\n", res1);    /* Non-compliant */
    printf("%s\n", copy);    /* Compliant */
    printf("%s\n", res2);    /* Compliant */
}
```

In this example:

- The first `printf` statement is non-compliant because the pointer returned by `setlocale` is used following a subsequent call to it when `res2` is assigned.

- The second `printf` statement is compliant because the copy operation performed by `strcpy` is made before a subsequent call to `setlocale` function is made.
- The third `printf` statement is compliant because there is no subsequent call to the `setlocale` function is made before use.

## Check Information

**Group:** Standard libraries
**Category:** Mandatory
**AGC Category:** Mandatory

# Version History

**Introduced in R2017a**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 21.3

The memory allocation and deallocation functions of `<stdlib.h>` shall not be used

## Description

### Rule Definition

*The memory allocation and deallocation functions of `<stdlib.h>` shall not be used.*

### Rationale

Using memory allocation and deallocation routines can cause undefined behavior. For instance:

- You free memory that you had not allocated dynamically.
- You use a pointer that points to a freed memory location.

### Polyspace Implementation

The checker flags uses of the `calloc`, `malloc`, `realloc`, `aligned_alloc` and `free` functions.

If you define macros with the same names as these dynamic heap memory allocation functions, and you expand the macros in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

### Additional Message in Report

- The macro <name> shall not be used.
- Identifier XX should not be used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `malloc`, `calloc`, `realloc` and `free`

```
#include <stdlib.h>

static int foo(void);

typedef struct struct_1 {
    int a;
    char c;
} S_1;

static int foo(void) {

    S_1 * ad_1;
    int  * ad_2;
    int  * ad_3;
    ad_1 = (S_1*)calloc(100U, sizeof(S_1));        /* Non-compliant */
    ad_2 = malloc(100U * sizeof(int));             /* Non-compliant */
    ad_3 = realloc(ad_3, 60U * sizeof(long));      /* Non-compliant */

    free(ad_1);                                    /* Non-compliant */
    free(ad_2);                                    /* Non-compliant */
    free(ad_3);                                    /* Non-compliant */

    return 1;
}
```

In this example, the rule is violated when the functions `malloc`, `calloc`, `realloc` and `free` are used.

## Check Information

**Group:** Standard Libraries
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`MISRA C:2012 Rule 18.7` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 21.4

The standard header file <setjmp.h> shall not be used

## Description

### Rule Definition

*The standard header file <setjmp.h> shall not be used*.

### Rationale

Using `setjmp` and `longjmp`, you can bypass normal function call mechanisms and cause undefined behavior.

### Polyspace Implementation

If the `longjmp` function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

### Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Standard Libraries
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 21.5

The standard header file <signal.h> shall not be used

## Description

### Rule Definition

*The standard header file <signal.h> shall not be used*.

### Rationale

Using signal handling functions can cause implementation-defined and undefined behavior.

### Polyspace Implementation

If the signal function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

### Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Standard Libraries
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 21.6

The Standard Library input/output functions shall not be used

## Description

### Rule Definition

*The Standard Library input/output functions shall not be used.*

### Rationale

This rule applies to the functions that are provided by `<stdio.h>` and in C99, their character-wide equivalents provided by `<wchar.h>`. Using these functions can cause unspecified, undefined and implementation-defined behavior.

### Polyspace Implementation

If the Standard Library function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

### Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Standard Libraries
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 21.7

The Standard Library functions `atof`, `atoi`, `atol`, and `atoll` functions of `<stdlib.h>` shall not be used

## Description

### Rule Definition

*The Standard Library functions `atof`, `atoi`, `atol`, and `atoll` functions of `<stdlib.h>` shall not be used.*

### Rationale

When a string cannot be converted, the behavior of these functions can be undefined.

### Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

### Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Standard Libraries
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 21.8

The Standard Library termination functions of `<stdlib.h>` shall not be used

## Description

### Rule Definition

The Standard Library termination functions of `<stdlib.h>` shall not be used.

### Rationale

Using these functions can cause undefined and implementation-defined behaviors.

### Polyspace Implementation

Polyspace flags the use of the `abort`, `exit`, `_Exit`, or `quick_exit` functions that are defined in `<stdlib.h>`.

If these functions are user-defined, Polyspace does not flag them.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Unsafe Termination Functions

```
#include<stdlib.h>

void foo(){
     puts("pushed");
    //...
    _Exit(-1);//Noncompliant
}
void bar(){
    puts("pushed");
    //...
    abort();//Noncompliant
}
void foobar(){
    puts("pushed");
    //...
    quick_exit(-1);//Noncompliant
}
```

In this example, unsafe termination functions are invoked to terminate the program. These functions might not perform the essential cleanup operations. For instance, the data pushed to the output stream might become lost because the program is terminated before the streams are closed. Polyspace flags the use of such unsafe termination programs.

## Check Information

**Group:** Standard Libraries
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 21.9

The Standard Library library functions `bsearch` and `qsort` of `<stdlib.h>` shall not be used

## Description

### Rule Definition

*The library functions `bsearch` and `qsort` of `<stdlib.h>` shall not be used.*

### Rationale

The comparison function in these library functions can behave inconsistently when the elements being compared are equal. Also, the implementation of `qsort` can be recursive and place unknown demands on the call stack.

### Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

### Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Standard Libraries
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 21.10

The Standard Library time and date functions shall not be used

## Description

### Rule Definition

*The Standard Library time and date functions shall not be used*.

### Rationale

Using these functions can cause unspecified, undefined and implementation-defined behavior.

### Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

### Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Standard Libraries
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 21.11

The standard header file <tgmath.h> shall not be used

## Description

### Rule Definition

*The standard header file <tgmath.h> shall not be used*.

### Rationale

Using the facilities of this header file can cause undefined behavior.

### Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

### Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Function in `tgmath.h`

```
#include <tgmath.h>//Noncompliant

float f1,res;


void func(void) {
    res = sqrt(f1); /* Non-compliant */
}
```

In this example, the rule is violated when the `sqrt` macro defined in `tgmath.h` is used.

#### Correction — Use Appropriate Function in `math.h`

For this example, one possible correction is to use the function `sqrtf` defined in `math.h` for `float` arguments.

```
#include <math.h>

float f1, res;
```

```
void func(void) {
 res = sqrtf(f1);
}
```

### Check Information

**Group:** Standard Libraries
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2014b**

### See Also

`Check MISRA C:2012 (-misra3)`

#### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 21.12

The exception handling features of `<fenv.h>` should not be used

## Description

### Rule Definition

*The exception handling features of* `<fenv.h>` *should not be used*.

### Rationale

In some cases, the values of the floating-point status flags are unspecified. Attempts to access them can cause undefined behavior.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Features in `<fenv.h>`

```
#include <fenv.h>

void func(float x, float y) {
    float z;

    feclearexcept(FE_DIVBYZERO);              /* Non-compliant */
    z = x/y;

    if(fetestexcept (FE_DIVBYZERO))  {        /* Non-compliant */
    }
    else {
#pragma STDC FENV_ACCESS ON
        z=x*y;
        if(z>x) {
#pragma STDC FENV_ACCESS OFF
            if(fetestexcept (FE_OVERFLOW)) {  /* Non-compliant */
            }
        }
    }
}
```

In this example, the rule is violated when the identifiers `feclearexcept` and `fetestexcept`, and the macros `FE_DIVBYZERO` and `FE_OVERFLOW` are used.

## Check Information

**Group:** Standard libraries
**Category:** Advisory
**AGC Category:** Advisory

## Version History

**Introduced in R2015b**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 21.13

Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF

## Description

### Rule Definition

*Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

Functions in <ctype.h> have a well-defined behavior only for int arguments whose value is within the range of unsigned char or the negative value equivalent of EOF. The use of other values results in undefined behavior.

### Polyspace Implementation

Polyspace considers that the negative value equivalent of EOF is -1 and does not raise a violation if you pass -1 as argument to a function in ctype.h.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Invalid Arguments for Functions from <ctype.h>

```
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <ctype.h>

bool f(uint8_t a)
{
    return (isdigit((int32_t) a)           /* Compliant     */
            &&  isalpha((int32_t) 'b')      /* Compliant     */
            &&  islower(EOF)                /* Compliant     */
            &&  isalpha(256));              /* Non-compliant */
}
```

In this example, the rule is violated when 256, which is an neither an unsigned char or the value EOF, is passed as an input argument to the isalpha function.

**Note** The int casts in the above example are required to comply with Rule 10.3 on page 21-146.

## Check Information

**Group:** Standard libraries
**Category:** Mandatory
**AGC Category:** Mandatory

## Version History

**Introduced in R2017a**

## See Also

`MISRA C:2012 Rule 10.3` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 21.14

The Standard Library function `memcmp` shall not be used to compare null terminated strings

## Description

### Rule Definition

*The Standard Library function `memcmp` shall not be used to compare null terminated strings.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

If `memcmp` is used to compare two strings and the length of either string is less than the number of bytes compared, the strings can appear different even when they are logically the same. The characters after the null terminator are compared even though they do not form part of the string.

For instance:

```
memcmp(string1, string2, sizeof(string1))
```

can compare bytes after the null terminator if `string1` is longer than `string2`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Using `memcmp` for String Comparison

```
extern char buffer1[ 12 ];
extern char buffer2[ 12 ];
void f1(void)
{
    (void) strcpy(buffer1, "abc");
    (void) strcpy(buffer2, "abc");

    if (memcmp(buffer1,    /* Non-compliant */
               buffer2,
               sizeof(buffer1)) != 0) {

    }
}
```

In this example, the comparison in the `if` statement is noncompliant. The strings stored in `buffer1` and `buffer2` can be reported different, but this difference comes from uninitialized characters after the null terminators.

## Check Information

**Group:** Standard libraries
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2017a**

## See Also

`MISRA C:2012 Rule 21.15` | `MISRA C:2012 Rule 21.16` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 21.15

The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types

## Description

### Rule Definition

*The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

The functions

```
memcpy( arg1, arg2, num_bytes );
memmove( arg1, arg2, num_bytes );
memcmp( arg1, arg2, num_bytes );
```

perform a byte-by-byte copy, move or comparison between the memory locations that `arg1` and `arg2` point to. A byte-by-byte copy, move or comparison is meaningful only if `arg1` and `arg2` have compatible types.

Using pointers to different data types for `arg1` and `arg2` typically indicates a coding error.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Incompatible Argument Types for `memcpy`

```
#include <stdint.h>

void f ( uint8_t s1[ 8 ], uint16_t s2[ 8 ] )
{
    ( void ) memcpy ( s1, s2, 8 ); /* Non-compliant */
}
```

In this example, `s1` and `s2` are pointers to different data types. The `memcpy` statement copies eight bytes from one buffer to another.

Eight bytes represent the entire span of the buffer that `s1` points to, but only part of the buffer that `s2` points to. Therefore, the `memcpy` statement copies only part of `s2` to `s1`, which might be unintended.

## Check Information

**Group:** Standard libraries
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2017a**

## See Also

`MISRA C:2012 Rule 21.14` | `MISRA C:2012 Rule 21.16` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 21.16

The pointer arguments to the Standard Library function memcmp shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type

## Description

### Rule Definition

*The pointer arguments to the Standard Library function* memcmp *shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

The Standard Library function

```
memcmp ( lhs, rhs, num );
```

performs a byte-by-byte comparison of the first num bytes of the two objects that lhs and rhs point to.

Do not use memcmp for a byte-by-byte comparison of the following.

| Type | Rationale |
|---|---|
| Structures | If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. The content of these extra padding bytes is meaningless. If you perform a byte-by-byte comparison of structures with memcmp, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value. |
| Objects with essentially floating type | The same floating point value can be stored using different representations. If you perform a byte-by-byte comparison of two variables with memcmp, you can reach the false conclusion that the variables are unequal even when they have the same value. The reason is that the values are stored using two different representations. |
| Essentially char arrays | Essentially char arrays are typically used to store strings. In strings, the content in bytes after the null terminator is meaningless. If you perform a byte-by-byte comparison of two strings with memcmp, you might reach the false conclusion that two strings are not equal, even if the bytes before the null terminator store the same value. |

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Using memcmp for Comparison of Structures, Unions, and *essentially char* Arrays

```
#include <stdbool.h>
#include <stdint.h>

struct S {
//...
};

bool f1(struct S* s1, struct S* s2)
{
    return (memcmp(s1, s2, sizeof(struct S)) != 0); /* Non-compliant */
}

union U {
    uint32_t range;
    uint32_t height;
};
bool f2(union U* u1, union U* u2)
{
    return (memcmp(u1, u2, sizeof(union U)) != 0); /* Non-compliant */
}

const char a[ 6 ] = "task";
bool f3(const char b[ 6 ])
{
    return (memcmp(a, b, 6) != 0); /* Non-compliant */
}
```

In this example:

- Structures s1 and s2 are compared in the bool_t f1 function. The return value of this function might indicate that s1 and s2 are different due to padding. This comparison is noncompliant.
- Unions u1 and u2 are compared in the bool_t f2 function. The return value of this function might indicate that u1 and u2 are the same due to unintentional comparison of u1.range and u2.height, or u1.height and u2.range. This comparison is noncompliant.
- Essentially char arrays a and b are compared in the bool_t f3 function. The return value of this function might incorrectly indicate that the strings are different because the length of a (four) is less than the number of bytes compared (six). This comparison is noncompliant.

## Check Information

**Group:** Standard libraries
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2017a**

## See Also

MISRA C:2012 Rule 21.14 | MISRA C:2012 Rule 21.15 | Check MISRA C:2012 (-misra3)

**Topics**

“Check for and Review Coding Standard Violations”
“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 21.17

Use of the string handling function from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters

## Description

### Rule Definition

*Use of the string handling function from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

Incorrect use of a string handling function might result in a read or write access beyond the bounds of the function arguments, resulting in undefined behavior.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Pointer Access Out of Bounds from `strcpy` Usage

```
#include <stdio.h>
#include<string.h>

char string[] = "Short";
void f1(const char* str)
{
    (void) strcpy(string, "Too long to fit");      /* Non-compliant */
    if (strlen(str) < (sizeof(string) - 1u)) {
        (void) strcpy(string, str);      /* Compliant */
    }
}

size_t f2(void)
{
    char text[ 5 ] = "Token";
    return strlen(text);    /* Non-compliant */
}
```

In this example:

- The first use of `strcpy` is noncompliant because it attempts to write beyond the end of its destination argument `string`.
- The second use of `strcpy` is compliant because it attempts to write to the destination argument `string` only if the source argument `str` fits.

- The use of `strlen` is noncompliant. `strlen` computes the length of a string up to the null terminator. The character array `text` has no null terminator.

## Check Information

**Group:** Standard libraries
**Category:** Mandatory
**AGC Category:** Mandatory

# Version History

**Introduced in R2017a**

## See Also

`MISRA C:2012 Rule 21.18` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 21.18

The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value

## Description

### Rule Definition

*The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

The value must be positive and not greater than the size of the smallest object passed by pointer to the function. For instance, suppose you use the `strncmp` function to compare two strings `lhs_string` and `rhs_string` as follows:

```
strncmp (lhs_string, rhs_string, num)
```

The third argument `num` must be positive and must not be greater than the size of `lhs_string` or `rhs_string`, whichever is smaller.

Otherwise, using the function can result in read or write access beyond the bounds of the function argument.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Incorrect `size_t` Argument for `memcmp`

```
#include <string.h>
char buf1[ 5 ] = "12345";
char buf2[ 10 ] = "1234567890";

void f(void)
{
    if (memcmp(buf1, buf2, 5) == 0) {    /* Compliant */

    }
    if (memcmp(buf1, buf2, 6) == 0) {    /* Non-compliant */

    }
}
```

In this example, the first `if` statement is compliant. The `size_t` argument is five, which is same as the size of the smaller string, `buf1`.

By the same reasoning, the second `if` statement is noncompliant.

## Check Information

**Group:** Standard libraries
**Category:** Mandatory
**AGC Category:** Mandatory

# Version History

**Introduced in R2017a**

## See Also

`MISRA C:2012 Rule 21.17` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 21.19

The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or `strerror` shall only be used as if they have pointer to `const`-qualified type

## Description

### Rule Definition

*The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or `strerror` shall only be used as if they have pointer to `const`-qualified type.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

The C99 Standard states that if the program modifies the structure pointed to by the value returned by `localeconv`, or the strings returned by `getenv`, `setlocale` or `strerro`, undefined behavior occurs. Treating the pointers returned by the various functions as if they were `const`-qualified allows an analysis tool to detect any attempt to modify an object through one of the pointers. Assigning the return values of the functions to `const`-qualified pointers results in the compiler issuing a diagnostic if an attempt is made to modify an object.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Returning Pointers from`setlocale` and `localeconv`

```
#include <locale.h>
#include <string.h>

void f1(void)
{
    char* s1 = setlocale(LC_ALL, 0);    /* Non-compliant */
    struct lconv* conv = localeconv();  /* Non-compliant */
    s1[ 1 ] = 'A'; /* Non-compliant. Undefined behavior */
    conv->decimal_point = "^"; /* Non-compliant. Undefined behavior */
}

void f2(void)
{
    char str[128];
    (void) strcpy(str, setlocale(LC_ALL, 0));     /* Compliant */
    const struct lconv* conv = localeconv();      /* Compliant */
    conv->decimal_point = "^";                    /* Non-compliant. Constraint violation */
}

void f3(void)
{
    const struct lconv* conv = localeconv();  /* Compliant */
    conv->grouping[ 2 ] = 'x';                /* Non-compliant */
}
```

In the above example:

- The usage of `setlocale` and `localeconv` in the function `f1` are non-compliant as the returned pointers are assigned to non-`const`—qualified pointers.

**Note** The usage of `setlocale` and `localeconv` above are not constraint violations and will therefore not be reported by a compiler. However, an analysis tool will be able to report a violation.

- The usage of `setlocale` in the function `f2` is compliant as `strcpy` takes a `const char *` as its second parameter. The usage of `localeconv` in the function `f2` is compliant as the returned pointers are assigned to a `const`-qualified pointer. Any attempt to modify an object through a pointer will be reported by a compiler or analysis tool as this is a constraint violation.
- The usage of a `const`-qualified pointer in the function `f3` gives compile time protection of the value returned by `localeconv` but the same is not true for the strings it references. Modification of these strings can be detected by an analysis tool.

## Check Information

**Group:** Standard libraries
**Category:** Mandatory
**AGC Category:** Mandatory

# Version History

**Introduced in R2017a**

## See Also

`MISRA C:2012 Rule 7.4` | `MISRA C:2012 Rule 11.8` | `MISRA C:2012 Rule 21.8` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 21.21

The Standard Library function `system` of `<stdlib.h>` shall not be used

## Description

### Rule Definition

*The Standard Library function `system` of `<stdlib.h>` shall not be used.*

This rule comes from MISRA C: 2012 Amendment 2.

### Rationale

If the argument of the `system` function is not sanitized, it can cause exploitable vulnerabilities. An attacker can execute arbitrary commands or read and modify data anywhere on the system.

### Polyspace Implementation

The checker flags uses of the Standard Library function `system`.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### `system()` Function Called

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};

void func_noncompliant(char *arg)
{
    char buf[SIZE512];
    int retval=sprintf(buf, "/usr/bin/any_cmd %s", arg);

    if (retval<=0 || retval>SIZE512){
      /* Handle error */
      abort();
    }
    /* Use of system() to pass any_cmd with
    unsanitized argument to command processor */

    if (system(buf) == -1) { //Noncompliant
    /* Handle error */
  }
}

void func_compliant(char *arg)
{
    char *const args[SIZE3] = {"any_cmd", arg, NULL};
    char  *const env[] = {NULL};

    /* Sanitize argument */

    /* Use execve() to execute any_cmd. */

    if (execve("/usr/bin/time", args, env) == -1) { //Compliant
      /* Handle error */
    }
}
```

In this example, in the `func_noncompliant` function, the `system` function passes its argument to the host environment for the command processor to execute. This code is vulnerable to an attack by command-injection.

In the compliant version of the same function, `func_compliant`, the argument of `any_cmd` is sanitized, and then passed to the `execve` function for execution. `exec`-family functions are not vulnerable to command-injection attacks.

## Check Information

**Group:** Standard Libraries
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2021a**

## See Also

`Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 22.1

All resources obtained dynamically by means of Standard Library functions shall be explicitly released

## Description

### Rule Definition

*All resources obtained dynamically by means of Standard Library functions shall be explicitly released*.

### Rationale

Resources are something that you must return to the system once you have used them. Examples include dynamically allocated memory and file descriptors.

If you do not release resources explicitly as soon as possible, then a failure can occur due to exhaustion of resources.

### Polyspace Implementation

The checker flags uses of:

- Memory-allocation functions such as `malloc` and `aligned_alloc` if the memory is not released.
- File opening functions such as `fopen` if the file is not closed.

You can check for this rule with a Bug Finder analysis only.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Dynamic Memory

```
#include<stdlib.h>

void performOperation(int);

int func1(int num)
{
    int* arr1 = (int*) malloc(num * sizeof(int));

    return 0; /* Non-compliant - memory allocated to arr1 is not released */
}

int func2(int num)
{
    int* arr2 = (int*) malloc(num * sizeof(int));

    free(arr2);
    return 0; /* Compliant - memory allocated to arr2 is released */
}
```

In this example, the rule is violated when memory dynamically allocated using the `malloc` function is not freed using the `free` function before the end of scope.

#### File Pointers

```
#include <stdio.h>
void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" );              /* Non-compliant */
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}

void func2( void ) {
    FILE *fp2;
    fp2 = fopen ( "data1.txt", "w" );
    fprintf ( fp2, "*" );
    fclose(fp2);

    fp2 = fopen ( "data2.txt", "w" );              /* Compliant */
    fprintf ( fp2, "!" );
    fclose ( fp2 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`. Therefore, the rule 22.1 is violated.

The rule is not violated in `func2` because file `data1.txt` is closed and the file pointer `fp2` is explicitly dissociated from `data1.txt` before it is reused.

## Check Information

**Group:** Resources
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2015b**

## See Also

`MISRA C:2012 Dir 4.13` | `MISRA C:2012 Rule 21.3` | `MISRA C:2012 Rule 21.6` | `Resource leak` | `Check MISRA C:2012 (-misra3)`

#### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 22.10

The value of `errno` shall only be tested when the last function to be called was an `errno`-setting function

## Description

### Rule Definition

*The value of `errno` shall only be tested when the last function to be called was an `errno`-setting function.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

Besides the `errno`-setting functions, the Standard does not enforce that other functions set `errno` on errors. Whether these functions set `errno` or not is implementation-dependent.

To detect errors, if you check `errno` alone, the validity of this check also becomes implementation-dependent. On implementations that do not require `errno` setting, even if you check `errno` alone, you can overlook error conditions.

For a list of `errno`-setting functions, see `MISRA C:2012 Rule 22.8`.

For information on how to detect errors, see the documentation for that specific function.

Typically, the functions return an out-of-band error indicator to indicate errors. For instance:

- `fopen` returns a null pointer if an error occurs.
- `signal` returns the `SIG_ERR` error indicator and sets `errno` to a positive value. Check `errno` only after you have checked the function return value.

### Polyspace Implementation

Polyspace raisea a violation of this rule when you check `errno` for error conditions in situations where checking `errno` does not guarantee the absence of errors. In some cases, checking `errno` can lead to false positives.

For instance, you check `errno` following calls to the functions:

- `fopen`: If you follow the ISO Standard, the function might not set `errno` on errors.
- `atof`: If you follow the ISO Standard, the function does not set `errno`.
- `signal`: The `errno` value indicates an error only if the function returns the `SIG_ERR` error indicator.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Incorrect Test of `errno`

```
#include <stddef.h>
#include <stdlib.h>

typedef double float64_t;

void f(void)
{
    float64_t f64;
    errno = 0;
    f64 = atof("A.12");
    if (0 == errno) { /* Non-compliant */
    }
    errno = 0;
    f64 = strtod("A.12", NULL);
    if (0 == errno) { /* Compliant */
    }
}
```

In this example:

- The first `if` statement is noncompliant because `atof` may or may not set `errno` when an error is detected. `f64` may not have a valid value within this `if` statement.
- The second `if` statement is compliant because `strtod` is an *`errno`-setting function*. `f64` will have a valid value within this `if` statement.

## Check Information

**Group:** Resources
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2017a**

## See Also

`MISRA C:2012 Rule 22.8` | `MISRA C:2012 Rule 22.9` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 22.2

A block of memory shall only be freed if it was allocated by means of a Standard Library function

## Description

### Rule Definition

*A block of memory shall only be freed if it was allocated by means of a Standard Library function.*

### Rationale

The Standard Library functions that allocate memory are `malloc`, `calloc` and `realloc`.

You free a block of memory when you pass its address to the `free` or `realloc` function. The following causes undefined behavior:

- You free a block of memory that you did not allocate.
- You free a block of memory that have already freed before.

### Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Memory Not Allocated Is Freed

```
#include <stdlib.h>

void func1(void) {
    int x=0;
    int *ptr=&x;

    free(ptr); /* Non-compliant: ptr is not dynamically allocated */

}
```

In this example, the rule is violated because the `free` function operates on a pointer that does not point to dynamically allocated memory.

### Memory Freed Twice

```
#include <stdlib.h>

void func(int arrSize) {
    int *ptr = (int*) malloc(arrSize* sizeof(int));
    free(ptr);   /* Block of memory freed once */
    free(ptr);   /* Non-compliant - Block of memory freed twice */
}
```

In this example, the rule is violated when the `free` function operates on `ptr` twice without a reallocation in between.

## Check Information

**Group:** Resources
**Category:** Mandatory
**AGC Category:** Mandatory

# Version History

**Introduced in R2015b**

## See Also

`MISRA C:2012 Dir 4.13` | `MISRA C:2012 Rule 21.3` | `Invalid free of pointer` | `Deallocation of previously deallocated pointer` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 22.3

The same file shall not be open for read and write access at the same time on different streams

## Description

### Rule Definition

*The same file shall not be open for read and write access at the same time on different streams.*

### Rationale

If a file is both written and read via different streams, the behavior can be undefined.

### Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Opening File That Is Open in Another Stream

```
#include <stdio.h>

void func(void) {
    FILE *fw = fopen("tmp.txt", "r+");
    FILE *fr = fopen("tmp.txt", "r");   /* Non-compliant: File open in stream fw*/
}
```

In this example, the rule is violated when the same file `tmp.txt` is opened in two streams. The `FILE` pointers `fw` and `fr` point to two different streams here.

## Check Information

**Group:** Resources
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2015b**

## See Also

`MISRA C:2012 Rule 21.6` | `Resource leak` | `Check MISRA C:2012 (-misra3)`

### Topics

“Check for and Review Coding Standard Violations”

“Software Quality Objective Subsets (C:2012)”

# MISRA C:2012 Rule 22.4

There shall be no attempt to write to a stream which has been opened as read-only

## Description

### Rule Definition

*There shall be no attempt to write to a stream which has been opened as read-only.*

### Rationale

The Standard does not specify the behavior if an attempt is made to write to a read-only stream.

### Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Writing to File Opened as Read-Only

```
#include <stdio.h>

void func1(void) {
    FILE *fp1 = fopen("tmp.txt", "r");
    (void) fprintf(fp1, "Some text"); /* Non-compliant: Read-only stream */
    (void) fclose(fp1);
}

void func2(void) {
    FILE *fp2 = fopen("tmp.txt", "r+");
    (void) fprintf(fp2, "Some text"); /* Compliant */
    (void) fclose(fp2);
}
```

In this example, the file stream associated with `fp1` is opened as read-only. The rule is violated when the stream is written.

## Check Information

**Group:** Resources
**Category:** Mandatory
**AGC Category:** Mandatory

# Version History

**Introduced in R2015b**

## See Also

`MISRA C:2012 Rule 21.6` | `Writing to read-only resource` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 22.5

A pointer to a FILE object shall not be dereferenced

## Description

### Rule Definition

*A pointer to a FILE object shall not be dereferenced*.

### Rationale

The Standard states that the address of a FILE object used to control a stream can be significant. Copying that object might not give the same behavior. This rule ensures that you cannot perform such a copy.

Directly manipulating a FILE object might be incompatible with its use as a stream designator.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### FILE* Pointer Dereferenced

```
#include <stdio.h>

void func(void) {
    FILE *pf1;
    FILE *pf2;
    FILE f3;

    pf2 = pf1;        /* Compliant */
    f3 = *pf2;        /* Non-compliant */
    pf2->_flags=0;    /* Non-compliant */
 }
```

In this example, the rule is violated when the FILE* pointer pf2 is dereferenced.

## Check Information

**Group:** Resources
**Category:** Mandatory
**AGC Category:** Mandatory

# Version History

**Introduced in R2015b**

## See Also

`MISRA C:2012 Rule 21.6 | Check MISRA C:2012 (-misra3)`

**Topics**

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 22.6

The value of a pointer to a `FILE` shall not be used after the associated stream has been closed

## Description

### Rule Definition

*The value of a pointer to a `FILE` shall not be used after the associated stream has been closed.*

### Rationale

The Standard states that the value of a `FILE*` pointer is indeterminate after you close the stream associated with it.

### Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use of `FILE` Pointer After Closing Stream

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fclose(fp);
        fprintf(fp,"text"); // Non-compliant
    }
}
```

In this example, the stream associated with the `FILE*` pointer `fp` is closed with the `fclose` function. The rule is violated `FILE*` pointer `fp` is used before the stream is re-opened.

## Check Information

**Group:** Resources
**Category:** Mandatory
**AGC Category:** Mandatory

# Version History

**Introduced in R2015b**

## See Also

`MISRA C:2012 Dir 4.13` | `MISRA C:2012 Rule 21.6` | `Use of previously closed resource` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 22.7

The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF

## Description

### Rule Definition

*The macro `EOF` shall only be compared with the unmodified return value from any Standard Library function capable of returning `EOF`.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

The `EOF` value may become indistinguishable from a valid character code if the value returned is converted to another type. In such cases, testing the converted value against `EOF` will not reliably identify if the end of the file has been reached or if an error has occurred.

### Troubleshooting

If you expect a rule violation but do not see it, refer to "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Possibly Misleading Results from Comparison with EOF

```
#include <stdio.h>
#include <stdint.h>

void f1(void)
{
    char ch;
    ch = (char) getchar();
    if (EOF != (int32_t) ch) {    /* Non-compliant */
    }
}

void f2(void)
{
    char ch;
    ch = (char) getchar();
    if (!feof(stdin)) {           /* Compliant */
    }
}

void f3(void)
{
    int32_t i_ch;
    i_ch = getchar();
    if (EOF != i_ch) {            /* Compliant */
```

```
        char ch;
        ch = (char) i_ch;
    }
}
```

In this example:

- The test in the `f1` function is non-compliant. It will not be reliable as the return value is cast to a narrower type before checking for `EOF`.
- The test in the `f2` function is compliant. It shows how *`feof()`* can be used to check for `EOF` when the return value from *`getchar()`* has been subjected to type conversion.
- The test in the `f3` function is compliant. It is reliable as the unconverted return value is used when checking for `EOF`.

### Check Information

**Group:** Resources
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2017a**

### See Also

`Check MISRA C:2012 (-misra3)`

#### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 22.8

The value of `errno` shall be set to zero prior to a call to an `errno`-setting-function

## Description

### Rule Definition

*The value of `errno` shall be set to zero prior to a call to an `errno`-setting-function.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

If an error occurs during a call to an `errno`-setting-function, the function writes a nonzero value to `errno`. Otherwise, `errno` is not modified.

If you do not explicitly set `errno` to zero before a function call, it can contain values from a previous call. Checking `errno` for nonzero values after the function call can give the false impression that an error occurred.

`Errno`-setting functions include:

- `ftell`, `fgetpos`, `fgetwc` and related functions.
- `strtoimax`, `strtol` and related functions.

  The wide-character equivalents such as `wcstoimax` and `wcstol` are also covered.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### `errno` Not Reset Before Use

```
#include <stdlib.h>
#include <errno.h>

double val = 0.0;

void f ( void )
{
    val = strtod("1.0",NULL);/*errno is not checked*/
    if ( 0 == errno ) /* Non-compliant*/
    {
        val = strtod("1.0",NULL); /* Compliant - case 1*/
        if ( 0 == errno ) /* Check errno for nonzero values */
        {
        }
    }
    else
    {
        errno = 0;
        val = strtod("1.0",NULL); /* Compliant - case 2*/
        if ( 0 == errno ) /* Check errno for nonzero values */
        {
        }
    }
}
```

In this example, the rule is violated when `strtod` is called but `errno` is not reset prior to the call.

The rule is not violated in the following cases:

- Case 1: `errno` is compared against zero and then `strtod` is called in the `if( 0 == errno )` branch.
- Case 2: `errno` is explicitly set to zero and then `strtod` is called.

## Check Information

**Group:** Resources
**Category:** Required
**AGC Category:** Required

# Version History

**Introduced in R2017a**

## See Also

`MISRA C:2012 Rule 22.9` | `MISRA C:2012 Rule 22.10` | `Check MISRA C:2012 (-misra3)`

### Topics

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

# MISRA C:2012 Rule 22.9

The value of `errno` shall be tested against zero after calling an `errno`-setting function

## Description

### Rule Definition

*The value of `errno` shall be tested against zero after calling an `errno`-setting function.*

This rule comes from MISRA C: 2012 Amendment 1.

### Rationale

If an error occurs during a call to an `errno`-setting-function, the function writes a nonzero value to `errno`. Otherwise, `errno` is not modified.

When `errno` is nonzero, the function return value is not likely to be correct. Before using this return value, you must test `errno` for nonzero values.

`Errno`-setting functions include:

- `ftell`, `fgetpos`, `fgetwc` and related functions.
- `strtoimax`, `strtol` and related functions.

  The wide-character equivalents such as `wcstoimax` and `wcstol` are also covered.

### Troubleshooting

If you expect a rule violation but do not see it, refer to “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### `errno` Not Tested After Function Call

```
#include <stdlib.h>
#include <errno.h>

void func(void);
double val = 0.0;

void f1 ( void )
{
  errno = 0;
  val = strtod ( "1.0", NULL ); /* Non-compliant */
  func ();

  if ( 0 != errno )
    {
    }

  errno = 0;
  val = strtod ( "1.0", NULL ); /* Compliant */
  if ( 0 == errno )
    {
      func();
    }
}
```

In this example, the rule is violated when `errno` is not checked immediately after the first call to `strtod`. Instead, a second function `func` is called. `func` might use the value in the global variable `val`. The value can be incorrect if an error has occurred during the call to `strtod`.

The rule is not violated when `errno` is checked before operations that potentially use the return value of `strtod`.

### Check Information

**Group:** Resources
**Category:** Required
**AGC Category:** Required

## Version History

**Introduced in R2017a**

### See Also

`MISRA C:2012 Rule 22.8` | `MISRA C:2012 Rule 22.10` | `Check MISRA C:2012 (-misra3)`

**Topics**

"Check for and Review Coding Standard Violations"
"Software Quality Objective Subsets (C:2012)"

22

# CERT C Rules and Recommendations

## Acknowledgement

This software has been created by MathWorks incorporating portions of: the “SEI CERT-C Website,” © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, ”SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition,” © 2016 Carnegie Mellon University, and “SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition” © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.

ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.

# CERT C: Rule PRE30-C

Do not create a universal character name through concatenation

## Description

### Rule Definition

*Do not create a universal character name through concatenation.*

### Polyspace Implementation

The rule checker checks for **Universal character name from token concatenation**.

## Examples

### Universal character name from token concatenation

#### Issue

**Universal character name from token concatenation** occurs when two preprocessing tokens joined with a ## operator create a universal character name. A universal character name begins with `\u` or `\U` followed by hexadecimal digits. It represents a character not found in the basic character set.

For instance, you form the character `\u0401` by joining two tokens:

```
#define assign(uc1, uc2, val) uc1##uc2 = val
...
assign(\u04, 01, 4);
```

#### Risk

The C11 Standard (Sec. 5.1.1.2) states that if a universal character name is formed by token concatenation, the behavior is undefined.

#### Fix

Use the universal character name directly instead of producing it through token concatenation.

#### Example - Universal Character Name from Token Concatenation

```
#define assign(uc1, uc2, val) uc1##uc2 = val

int func(void) {
    int \u0401 = 0;
    assign(\u04, 01, 4);  //Noncompliant
    return \u0401;
}
```

In this example, the `assign` macro, when expanded, joins the two tokens `\u04` and `01` to form the universal character name `\u0401`.

#### Correction — Use Universal Character Name Directly

One possible correction is to use the universal character name `\u0401` directly. The correction redefines the `assign` macro so that it does not join tokens.

```
#define assign(ucn, val) ucn = val

int func(void) {
    int \u0401 = 0;
    assign(\u0401, 4);
    return \u0401;
}
```

## Check Information

**Group:** Rule 01. Preprocessor (PRE)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

PRE30-C

# CERT C: Rule PRE31-C

Avoid side effects in arguments to unsafe macros

## Description

### Rule Definition

*Avoid side effects in arguments to unsafe macros.*

### Polyspace Implementation

The rule checker checks for **Side effect in arguments to unsafe macro**.

## Examples

### Side effect in arguments to unsafe macro

#### Issue

**Side effect in arguments to unsafe macro** occurs when you call an unsafe macro with an expression that has a side effect.

- *Unsafe macro*: When expanded, an unsafe macro evaluates its arguments multiple times or does not evaluate its argument at all.

  For instance, the `ABS` macro evaluates its argument `x` twice.

  ```
  #define ABS(x) (((x) < 0) ? -(x) : (x))
  ```
- *Side effect*: When evaluated, an expression with a side effect modifies at least one of the variables in the expression.

  For instance, `++n` modifies `n`, but `n+1` does not modify `n`.

  The checker does not consider side effects in nested macros. The checker also does not consider function calls or volatile variable access as side effects.

#### Risk

If you call an unsafe macro with an expression that has a side effect, the expression is evaluated multiple times or not evaluated at all. The side effect can occur multiple times or not occur at all, causing unexpected behavior.

For instance, in the call `MACRO(++n)`, you expect only one increment of the variable `n`. If `MACRO` is an unsafe macro, the increment happens more than once or does not happen at all.

The checker flags expressions with side effects in the `assert` macro because the `assert` macro is disabled in non-debug mode. To compile in non-debug mode, you define the `NDEBUG` macro during compilation. For instance, in GCC, you use the flag `-DNDEBUG`.

#### Fix

Evaluate the expression with a side effect in a separate statement, and then use the result as a macro argument.

For instance, instead of:

```
MACRO(++n);
```

perform the operation in two steps:

```
++n;
MACRO(n);
```

Alternatively, use an inline function instead of a macro. Pass the expression with side effect as argument to the inline function.

The checker considers modifications of a local variable defined only in the block scope of a macro body as a side effect. This defect cannot happen since the variable is visible only in the macro body. If you see a defect of this kind, ignore the defect.

**Example - Macro Argument with Side Effects**

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  int m = ABS(++n); //Noncompliant

  /* ... */
}
```

In this example, the ABS macro evaluates its argument twice. The second evaluation can result in an unintended increment.

**Correction — Separate Evaluation of Expression from Macro Usage**

One possible correction is to first perform the increment, and then pass the result to the macro.

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  ++n;
  int m = ABS(n);

  /* ... */
}
```

**Correction — Evaluate Expression in Inline Function**

Another possible correction is to evaluate the expression in an inline function.

```
static inline int iabs(int x) {
  return (((x) < 0) ? -(x) : (x));
}

void func(int n) {
  /* Validate that n is within the desired range */

int m = iabs(++n);

  /* ... */
}
```

## Check Information

**Group:** Rule 01. Preprocessor (PRE)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

PRE31-C

# CERT C: Rule PRE32-C

Do not use preprocessor directives in invocations of function-like macros

## Description

### Rule Definition

*Do not use preprocessor directives in invocations of function-like macros.*

### Polyspace Implementation

The rule checker checks for **Preprocessor directive in macro argument**.

## Examples

### Preprocessor directive in macro argument

#### Issue

**Preprocessor directive in macro argument** occurs when you use a preprocessor directive in the argument to a function-like macro or a function that might be implemented as a function-like macro.

For instance, a `#ifdef` statement occurs in the argument to the function like macro `foo()`:

```
#define foo(X) //...
foo(
    #ifdef A
    "A"
    #else
    "B"
    #endif
  );
```

The checker reports a violation on this kind of usage of `#ifdef` statements. Violations are also reported when `#ifdef` statements are used as arguments of library functions that might be implemented as macros. Examples of library functions that might be implemented as macros include `assert()`, `memcpy()`, and `printf()`.

#### Risk

During preprocessing, a function-like macro call is replaced by the macro body and the parameters are replaced by the arguments to the macro call (argument substitution). Suppose a macro `min()` is defined as follows.

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you call `min(1,2)`, it is replaced by the body `((X) < (Y) ? (X) : (Y))`. `X` and `Y` are replaced by 1 and 2.

According to the C11 Standard (Sec. 6.10.3), if the list of arguments to a function-like macro itself has preprocessing directives, the argument substitution during preprocessing is undefined.

#### Fix

To ensure that the argument substitution happens in an unambiguous manner, use the preprocessor directives outside the function-like macro.

For instance, to execute `foo` with different arguments based on a `#ifdef` directive, call `foo` multiple times within the `#ifdef` directive branches.

```
#ifdef A
    foo("A");
#else
    foo("B")
#endif
```

#### Example - Directives in Function-Like Macros

In this example, the preprocessor directives `#ifdef` and `#endif` occur in the argument to the function-like macro `print()`. Polyspace reports violations on the preprocessor directives in macro arguments.

```
#include <stdio.h>

#define print(A) printf(#A)

void func(void) {
    print(
#ifdef SW //Noncompliant
          "Message 1"
#else
          "Message 2"
#endif //Noncompliant
         );
}
```

#### Correction — Use Directives Outside Macro

One possible correction is to use the function-like macro multiple times in the branches of the `#ifdef` directive.

```
#include <stdio.h>

#define print(A) printf(#A)

void func(void) {
#ifdef SW
        print("Message 1");
#else
        print("Message 2");
#endif
}
```

## Check Information

**Group:** Rule 01. Preprocessor (PRE)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

PRE32-C

# CERT C: Rule DCL30-C

Declare objects with appropriate storage durations

## Description

### Rule Definition

*Declare objects with appropriate storage durations.*

### Polyspace Implementation

The rule checker checks for **Pointer or reference to stack variable leaving scope**.

## Examples

### Pointer or reference to stack variable leaving scope

#### Issue

**Pointer or reference to stack variable leaving scope** occurs when a pointer or reference to a local variable leaves the scope of the variable. For instance:

- A function returns a pointer to a local variable.
- A function performs the assignment `globPtr = &locVar`. `globPtr` is a global pointer variable and `locVar` is a local variable.
- A function performs the assignment `*paramPtr = &locVar`. `paramPtr` is a function parameter that is, for instance, an `int**` pointer and `locVar` is a local `int` variable.
- A C++ method performs the assignment `memPtr = &locVar`. `memPtr` is a pointer data member of the class the method belongs to. `locVar` is a variable local to the method.

The defect also applies to memory allocated using the `alloca` function. The defect does not apply to static, local variables. Polyspace assumes that the local objects within a function definition are in the same scope.

#### Risk

Local variables are allocated an address on the stack. Once the scope of a local variable ends, this address is available for reuse. Using this address to access the local variable value outside the variable scope can cause unexpected behavior.

If a pointer to a local variable leaves the scope of the variable, Polyspace Bug Finder highlights the defect. The defect appears even if you do not use the address stored in the pointer. For maintainable code, it is a good practice to not allow the pointer to leave the variable scope. Even if you do not use the address in the pointer now, someone else using your function can use the address, causing undefined behavior.

#### Fix

Do not allow a pointer or reference to a local variable to leave the variable scope.

#### Example - Pointer to Local Variable Returned from Function

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0; //Noncompliant
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In this example, `func1` returns a pointer to local variable `ret`.

In `main`, `ptr` points to the address of the local variable. When `ptr` is accessed in `func2`, the access is illegal because the scope of `ret` is limited to `func1`,

## Check Information

**Group:** Rule 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL30-C

# CERT C: Rule DCL31-C

Declare identifiers before using them

## Description

### Rule Definition

*Declare identifiers before using them.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Types not explicitly specified**.
- **Implicit function declaration**.

This violation is reported on each use of a noncompliant identifier.

## Examples

### Types not explicitly specified

#### Issue

The rule checker flags situations where a function parameter or return type is not explicitly specified. To enable checking of this rule, use the value `c90` for the option `C standard version (-c-version)`.

#### Risk

In some circumstances, you can omit types from the C90 standard. In those cases, the `int` type is implicitly specified. However, the omission of an explicit type can lead to confusion. For example, in the declaration `extern void foo (char c, const k);`, the type of `k` is `const int`, but you might expect `const char`.

You might be using an implicit type in:

- Object declarations
- Parameter declarations
- Member declarations
- `typedef` declarations
- Function return types

#### Example - Implicit Types

```
static foo(int a);  /* Non compliant */
static void bar(void);      /* Compliant */
```

In this example, the rule is violated because the return type of `foo` is implicit.

### Implicit function declaration

**Issue**

The issue occurs when you call a function before you declare or define it.

**Risk**

An implicit declaration occurs when you call a function before declaring or defining it. When you declare a function explicitly before calling it, the compiler can match the argument and return types with the parameter types in the declaration. If an implicit declaration occurs, the compiler makes assumptions about the argument and return types. For instance, it assumes a return type of `int`. The assumptions might not agree with what you expect and cause undesired type conversions.

**Example - Function Not Declared Before Call**

```
#include <math.h>

extern double power3 (double val, int exponent);
int getChoice(void);

double func() {
    double res;
    int ch = getChoice();
    if(ch == 0) {
        res = power(2.0, 10);    /* Non-compliant */
    }
    else if( ch==1) {
        res = power2(2.0, 10);   /* Non-compliant */
    }
    else {
        res = power3(2.0, 10);   /* Compliant */
        return res;
    }
}

double power2 (double val, int exponent) {
    return (pow(val, exponent));
}
```

In this example, the rule is violated when a function that is not declared is called in the code. Even if a function definition exists later in the code, the rule violation occurs.

The rule is not violated when the function is declared before it is called in the code. If the function definition exists in another file and is available only during the link phase, you can declare the function in one of the following ways:

- Declare the function with the `extern` keyword in the current file.
- Declare the function in a header file and include the header file in the current file.

## Check Information

**Group:** Rule 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL31-C

# CERT C: Rule DCL36-C

Do not declare an identifier with conflicting linkage classifications

## Description

### Rule Definition

*Do not declare an identifier with conflicting linkage classifications.*

### Polyspace Implementation

The rule checker checks for **Inconsistent use of static and extern in object declarations**.

## Examples

### Inconsistent use of `static` and `extern` in object declarations

#### Issue

The issue occurs when you do not use the `static` storage class specifier consistently in all declarations of object and functions that have internal linkage.

The rule checker detects situations where:

- The same object is declared multiple times with different storage specifiers.
- The same function is declared and defined with different storage specifiers.

#### Risk

If you do not use the `static` specifier consistently in all declarations of objects with internal linkage, you might declare the same object with external and internal linkage.

In this situation, the linkage follows the earlier specification that is visible (C99 Standard, Section 6.2.2). For instance, if the earlier specification indicates internal linkage, the object has internal linkage even though the latter specification indicates external linkage. If you notice the latter specification alone, you might expect otherwise.

#### Example - Linkage Conflict Between Variable Declarations

```
static int foo = 0;
extern int foo;         /* Non-compliant */

extern int hhh;
static int hhh;         /* Non-compliant */
```

In this example, the first line defines `foo` with internal linkage. The first line is compliant because the example uses the `static` keyword. The second line does not use `static` in the declaration, so the declaration is noncompliant. By comparison, the third line declares `hhh` with an `extern` keyword creating external linkage. The fourth line declares `hhh` with internal linkage, but this declaration conflicts with the first declaration of `hhh`.

#### Correction — Consistent `static` and `extern` Use

One possible correction is to use `static` and `extern` consistently:

```
static int foo = 0;
static int foo;

extern int hhh;
extern int hhh;
```

#### Example - Linkage Conflict Between Function Declaration and Definition

```
static int fee(void);  /* Compliant - declaration: internal linkage */
int fee(void){         /* Non-compliant */
  return 1;
}

static int ggg(void);  /* Compliant - declaration: internal linkage */
extern int ggg(void){  /* Non-compliant */
  return 1;
}
```

This example shows two internal linkage violations. Because `fee` and `ggg` have internal linkage, you must use a `static` class specifier to be compliant with MISRA.

## Check Information

**Group:** Rule 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL36-C

# CERT C: Rule DCL37-C

Do not declare or define a reserved identifier

## Description

### Rule Definition

*Do not declare or define a reserved identifier.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Defining and undefining reserved identifiers or macros**.
- **Declaring a reserved identifier or macro name**.

If your code declares or defines a reserved identifier:

- If there is an explicit definition, Polyspace flags it.
- If there is no explicit definition, the compiler makes an implicit definition when your code uses the symbol. Polyspace flags such a use.

This violation is reported once for each noncompliant identifier.

## Examples

### Defining and undefining reserved identifiers or macros

#### Issue

The issue occurs when you use `#define` and `#undef` on a reserved identifier or reserved macro name.

#### Risk

Reserved identifiers and reserved macro names are intended for use by the implementation. Removing or changing the meaning of a reserved macro can result in undefined behavior. This rule applies to the following:

- Identifiers or macro names beginning with an underscore
- Identifiers in file scope described in the C Standard Library (ISO/IEC 9899:1999, Section 7, "Library")
- Macro names described in the C Standard Library as being defined in a standard header (ISO/IEC 9899:1999, Section 7, "Library").

#### Example - Defining or Undefining Reserved Identifiers

```
#undef __LINE__               /* Non-compliant - begins with _ */
#define _Guard_H 1            /* Non-compliant - begins with _ */
#undef _ BUILTIN_sqrt        /* Non-compliant - implementation may
                                * use _BUILTIN_sqrt for other purposes,
                               * e.g. generating a sqrt instruction */
#define defined               /* Non-compliant - reserved identifier */
#define errno my_errno        /* Non-compliant - library identifier */
#define isneg(x) ( (x) < 0 )  /* Compliant - rule doesn't include
                               * future library directions   */
```

#### Declaring a reserved identifier or macro name

**Issue**

The issue occurs when you declare a reserved identifier or macro name.

If you define a macro name that corresponds to a standard library macro, object, or function, Polyspace considers this a violation of the rule.

The rule considers tentative definitions as definitions.

**Risk**

The Standard allows implementations to treat reserved identifiers specially. If you reuse reserved identifiers, you can cause undefined behavior.

## Check Information

**Group:** Rule 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL37-C

# CERT C: Rule DCL38-C

Use the correct syntax when declaring a flexible array member

## Description

### Rule Definition

*Use the correct syntax when declaring a flexible array member.*

### Polyspace Implementation

The rule checker checks for **Incorrect syntax of flexible array member size**.

## Examples

### Incorrect syntax of flexible array member size

#### Issue

**Incorrect syntax of flexible array member size** occurs when you do not use the standard C syntax to define a structure with a flexible array member.

Since C99, you can define a flexible array member with an unspecified size. For instance, `desc` is a flexible array member in this example:

```
struct record {
    size_t len;
    double desc[];
};
```

Prior to C99, you might have used compiler-specific methods to define flexible arrays. For instance, you used arrays of size one or zero:

```
struct record {
    size_t len;
    double desc[0];
};
```

This usage is not compliant with the C standards following C99.

#### Risk

If you define flexible array members by using size zero or one, your implementation is compiler-dependent. For compilers that do not recognize the syntax, an `int` array of size one has buffer for one `int` variable. If you try to write beyond this buffer, you can run into issues stemming from array access out of bounds.

If you use the standard C syntax to define a flexible array member, your implementation is portable across all compilers conforming with the standard.

#### Fix

To implement a flexible array member in a structure, define an array of unspecified size. The structure must have one member besides the array and the array must be the last member of the structure.

#### Example - Flexible Array Member Defined with Size One

```
#include <stdlib.h>

struct flexArrayStruct {
  int num;
  int data[1]; //Noncompliant
};

unsigned int max_size = 100;

void func(unsigned int array_size) {
  if(array_size<= 0 || array_size > max_size)
      exit(1);
  /* Space is allocated for the struct */
  struct flexArrayStruct *structP
    = (struct flexArrayStruct *)
     malloc(sizeof(struct flexArrayStruct)
          + sizeof(int) * (array_size - 1));
  if (structP == NULL) {
    /* Handle malloc failure */
    exit(2);
  }

  structP->num = array_size;

  /*
   * Access data[] as if it had been allocated
   * as data[array_size].
   */
  for (unsigned int i = 0; i < array_size; ++i) {
    structP->data[i] = 1;
  }

  free(structP);
}
```

In this example, the flexible array member `data` is defined with a size value of one. Compilers that do not recognize this syntax treat `data` as a size-one array. The statement `structP->data[i] = 1;` can write to `data` beyond the first array member and cause out of bounds array issues.

#### Correction — Use Standard C Syntax to Define Flexible Array

Define flexible array members with unspecified size.

```
#include <stdlib.h>

struct flexArrayStruct{
  int num;
  int data[];
};
```

```
unsigned int max_size = 100;

void func(unsigned int array_size) {
  if(array_size<=0 || array_size > max_size)
      exit(1);

  /* Allocate space for structure */
  struct flexArrayStruct *structP
    = (struct flexArrayStruct *)
    malloc(sizeof(struct flexArrayStruct)
         + sizeof(int) * array_size);

  if (structP == NULL) {
    /* Handle malloc failure */
    exit(2);
  }

  structP->num = array_size;

  /*
   * Access data[] as if it had been allocated
   * as data[array_size].
   */
  for (unsigned int i = 0; i < array_size; ++i) {
    structP->data[i] = 1;
  }

  free(structP);
}
```

### Check Information

**Group:** Rule 02. Declarations and Initialization (DCL)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

DCL38-C

# CERT C: Rule DCL39-C

Avoid information leakage in structure padding

## Description

### Rule Definition

*Avoid information leakage in structure padding.*

### Polyspace Implementation

The rule checker checks for **Information leak via structure padding**.

## Examples

### Information leak via structure padding

#### Issue

**Information leak via structure padding** occurs when you do not initialize the padding data of a structure or union before passing it across a trust boundary. A compiler adds padding bytes to the structure or union to ensure a proper memory alignment of its members. The bit-fields of the storage units can also have padding bits.

**Information leak via structure padding** raises a defect when:

- You call an untrusted function with structure or union pointer type argument containing uninitialized padding data.

  All external functions are considered untrusted.
- You copy or assign a structure or union containing uninitialized padding data to an untrusted object.

  All external structure or union objects, the output parameters of all externally linked functions, and the return pointer of all external functions are considered untrusted objects.

#### Risk

The padding bytes of the passed structure or union might contain sensitive information that an untrusted source can access.

#### Fix

- Prevent the addition of padding bytes for memory alignment by using the `pack` pragma or attribute supported by your compiler.
- Explicitly declare and initialize padding bytes as fields within the structure or union.
- Explicitly declare and initialize bit-fields corresponding to padding bits, even if you use the `pack` pragma or attribute supported by your compiler.

#### Example - Structure with Padding Bytes Passed to External Function

```
#include <stddef.h>
#include <stdlib.h>
```

```
#include <string.h>

typedef struct s_padding
{
 /* Padding bytes may be introduced between
 * 'char c' and 'int i'
 */
        char c;
    int i;

/*Padding bits may be introduced around the bit-fields
* even if you use "#pragma pack" (Windows) or
* __attribute__((__packed__)) (GNU)*/

    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;



/* External function */
extern void copy_object(void *out, void *in, size_t s);

void func(void *out_buffer)
{
/*Padding bytes not initialized*/

    S_Padding s = {'A', 10, 1, 3, {}};
/*Structure passed to external function*/

    copy_object((void *)out_buffer, (void *)&s, sizeof(s)); //Noncompliant
}

void main(void)
{
    S_Padding s1;
    func(&s1);
}
```

In this example, structure `s1` can have padding bytes between the `char c` and `int i` members. The bit-fields of the storage units of the structure can also contain padding bits. The content of the padding bytes and bits is accessible to an untrusted source when `s1` is passed to `func`.

**Correction — Use pack Pragma to Prevent Padding Bytes**

One possible correction in Microsoft Visual Studio is to use `#pragma pack()` to prevent padding bytes between the structure members. To prevent padding bits in the bit-fields of `s1`, explicitly declare and initialize the bit-fields even if you use `#pragma pack()`.

```
 #include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define CHAR_BIT 8

#pragma pack(push, 1)
```

```
typedef struct s_padding
{
/*No Padding bytes when you use "#pragma pack" (Windows) or
* __attribute__((__packed__)) (GNU)*/
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
/* Padding bits explicitely declared */
    unsigned int bf_filler : sizeof(unsigned) * CHAR_BIT - 3;
    unsigned char buffer[20];
}

    S_Padding;

#pragma pack(pop)


/* External function */
extern void copy_object(void *out, void *in, size_t s);



void func(void *out_buffer)
{
    S_Padding s = {'A', 10, 1, 3, 0 /* padding bits */, {}};
    copy_object((void *)out_buffer, (void *)&s, sizeof(s));
}

void main(void)
{
    S_Padding s1;
    func(&s1);
}
```

## Check Information

**Group:** Rule 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C (-cert-c)

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

DCL39-C

# CERT C: Rule DCL40-C

Do not create incompatible declarations of the same function or object

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.

### Rule Definition

*Do not create incompatible declarations of the same function or object.*

### Polyspace Implementation

The rule checker checks for **Declaration mismatch**.

## Examples

### Declaration mismatch

#### Issue

**Declaration mismatch** occurs when a function or variable declaration does not match other instances of the function or variable.

#### Risk

When a mismatch occurs between two variable declarations in different compilation units, a typical linker follows an algorithm to pick one declaration for the variable. If you expect a variable declaration that is different from the one chosen by the linker, you can see unexpected results when the variable is used.

A similar issue can occur with mismatch in function declarations.

#### Fix

The fix depends on the type of declaration mismatch. If both declarations indeed refer to the same object, use the same declaration. If the declarations refer to different objects, change the names of the one of the variables. If you change a variable name, remember to make the change in all places that use the variable.

Sometimes, declaration mismatches can occur because the declarations are affected by previous preprocessing directives. For instance, a declaration occurs in a macro, and the macro is defined on one inclusion path but undefined in another. These declaration mismatches can be tricky to debug. Identify the divergence between the two inclusion paths and fix the conflicting macro definitions.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.

- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Inconsistent Declarations in Two Files

*file1.c*

```
int foo(void) {
    return 1;
}
```

*file2.c*

```
double foo(void); //Noncompliant

int bar(void) {
    return (int)foo();
}
```

In this example, *file1.c* declares `foo()` as returning an integer. In *file2.c*, `foo()` is declared as returning a double. This difference might cause a compilation failure. Polyspace raises a defect on the second instance of `foo` in *file2*.

#### Correction — Align the Function Return Values

One possible correction is to change the function declarations so that they match. In this example, by changing the declaration of `foo` in *file2.c* to match *file1.c*, the defect is fixed.

*file1.c*

```
int foo(void) {
    return 1;
}
```

*file2.c*

```
int foo(void);

int bar(void) {
    return foo();
}
```

#### Example - Inconsistent Structure Alignment

*test1.c*

```
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

*test2.c*

```
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square; //Noncompliant

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

*circle.h*

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;
```

*square.h*

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

In this example, a declaration mismatch defect is raised on `square` in *square.h* because Polyspace infers that `square` in *square.h* does not have the same alignment as `square` in *test2.c*. This error occurs because the `#pragma pack(1)` statement in *circle.h* declares specific alignment. In *test2.c*, *circle.h* is included before *square.h*. Therefore, the `#pragma pack(1)` statement from *circle.h* is not reset to the default alignment after the `aCircle` structure. Because of this omission, *test2.c* infers that the `aSquare square` structure also has an alignment of `1` byte. This defect might cause a compilation failure.

#### Correction — Close Packing Statements

One possible correction is to reset the structure alignment after the `aCircle` struct declaration. For the GNU or Microsoft Visual compilers, fix the defect by adding a `#pragma pack()` statement at the end of *circle.h*.

*test1.c*

```
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

*test2.c*

```
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

*circle.h*

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;

#pragma pack()
```

*square.h*

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

Other compilers require different `#pragma pack` syntax. For your syntax, see the documentation for your compiler.

#### Correction — Use the `Ignore pragma pack directives` Option

One possible correction is to add the `Ignore pragma pack directives` option to your Bug Finder analysis. If you want the structure alignment to change for each structure, and you do not want to see this **Declaration mismatch** defect, use this correction.

1. On the Configuration pane, select the **Advanced Settings** pane.
2. In the **Other** box, enter `-ignore-pragma-pack`.

3 Rerun your analysis.

The **Declaration mismatch** defect is resolved.

## Check Information

**Group:** Rule 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

DCL40-C

# CERT C: Rule DCL41-C

Do not declare variables inside a switch statement before the first case label

## Description

### Rule Definition

*Do not declare variables inside a switch statement before the first case label.*

### Polyspace Implementation

The rule checker checks for **variable declaration before first `case` label**.

## Examples

### Variable declaration before first case label

**Issue**

The issue occurs when you define a variable in a `switch` block before the first `case` label.

**Risk**

In a `switch` block, control jumps to one of the `case` labels or a `default` label, depending on the control expression of the `switch` statement. If you define a variable before the first `case` label, the compiler ignores the variable declaration. Read operations on this variable can lead to indeterminate values.

**Example — Noncompliant Variable Declaration**

```
void bar(int iTemp){
    //...
}
void foo(){
    //...
    int bFlag;
    //...
    switch(bFlag){
        int temp;  //Noncompliant
        bar (temp);
    case 0:
            //...
        break;
    case 1:
            //...
        break;

    }

}
```

In this example, the variable `temp` is declared before the first `case` label. The compiler ignores this declaration. If you perform a read operation on `temp` in one of the case blocks, the operation might lead to indeterminate values. Polyspace flags the variable declaration.

#### Example — Compliant switch Statement

To fix the preceding issue, move the variable declaration out of the `switch` statement.

```
void bar(int iTemp){
    //...
}
void foo(){
    //...
    int bFlag;
    //...
    int temp;//Compliant
    bar (temp);
    switch(bFlag){

    case 0://...
        break;
    case 1:
        break;
    }
}
```

## Check Information

**Group:** Rule 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL41-C

# CERT C: Rule EXP30-C

Do not depend on the order of evaluation for side effects

## Description

### Rule Definition

*Do not depend on the order of evaluation for side effects.*

### Polyspace Implementation

The rule checker checks for **Expression value depends on order of evaluation or of side effects**.

## Examples

### Expression value depends on order of evaluation or of side effects

**Issue**

The issue occurs when the value of an expression and its persistent side effects is not the same under all permitted evaluation orders.

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.
- The expression contains a single `volatile` object that occurs multiple times.
- The expression contains more than one `volatile` object.

Therefore, this rule forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation.

**Risk**

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

**Example - Variable Modified More Than Once in Expression**

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         /* Compliant */
    COPY_ELEMENT (i++);       /* Noncompliant  */
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

#### Example - Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
    f ( i++, i );                 /* Non-compliant */
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

#### Multiple `volatile` Objects in an Expression

```
volatile int a, b;
int mathOp(int x, int y);

int foo(void){
    int temp = mathOp(5,a) + mathOp(6,b);//Noncompliant
    return temp * mathOp(a,a);//Noncompliant
}
```

In this example, this rule is violated twice.

- The declaration of `temp` uses two `volatile` objects in the expression. Because the value of `volatile` objects might change at any time, the expression might evaluate to different values depending on the order of evaluation. Polyspace flags the second `volatile` object in the expression.
- The `return` statement uses the same `volatile` object twice. Because the expression might have different results depending on the order of evaluation, Polyspace raises this defect.

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP30-C

# CERT C: Rule EXP32-C

Do not access a volatile object through a nonvolatile reference

## Description

### Rule Definition

*Do not access a volatile object through a nonvolatile reference.*

### Polyspace Implementation

The rule checker checks for **Cast to pointer that removes volatile qualification**.

## Examples

### Cast to pointer that removes `volatile` qualification

**Issue**

Polyspace flags both implicit and explicit conversions that violate this rule.

**Risk**

This rule forbids casts from a pointer to a `volatile` object to a pointer that does not point to a `volatile` object. Such casts violate type qualification.

**Example - Casts That Remove Qualifiers**

```
void foo(void) {
    volatile unsigned short *pvi;    /* pointer to volatile  */
    unsigned short          *pi;

    pi  = (unsigned short *)  pvi;   /* Non-compliant */

}
```

In this example, the variable `pvi` has a `volatile` qualifier in its type. The rule is violated when the variable is cast to a type that does not have the `volatile` qualifier.

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP32-C

# CERT C: Rule EXP33-C

Do not read uninitialized memory

## Description

### Rule Definition

*Do not read uninitialized memory.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Non-initialized pointer**.
- **Non-initialized variable**.

### Extend Checker

You can extend the checker in the following ways:

- Polyspace does not flag passing pointers to noninitialized variables to functions. To detect noninitialized variables that are passed to functions by pointers, extend the checker by using the option `-code-behavior-specification`. See “Extend Checkers for Initialization to Check Function Arguments Passed by Pointers”.
- If a variable in your code is non-initialized only for certain system input values, you can see one possible combination of input values causing the defect. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Non-initialized pointer

#### Issue

**Non-initialized pointer** occurs when a pointer is not assigned an address before dereference.

#### Risk

Unless a pointer is explicitly assigned an address, it points to an unpredictable location.

#### Fix

The fix depends on the root cause of the defect. For instance, you assigned an address to the pointer but the assignment is unreachable.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below. It is a good practice to initialize a pointer to NULL when declaring the pointer.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Non-initialized pointer error**

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL) 
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j;                     //Noncompliant
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

**Correction — Initialize Pointer on Every Execution Path**

One possible correction is to assign an address to `pi` when `prev` is not `NULL`.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL) 
       {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
       }
    /* Fix: Initialize pi in branches of if statement  */
    else 
        pi = prev;


    *pi = j;

    return pi;
}
```

### Non-initialized variable

#### Issue

**Non-initialized variable** occurs when a variable is not initialized before its value is read.

#### Risk

Unless a variable is explicitly initialized, the variable value is unpredictable. You cannot rely on the variable having a specific value.

#### Fix

The fix depends on the root cause of the defect. For instance, you assigned a value to the variable but the assignment is unreachable or you assigned a value to the variable in one of two branches of a conditional statement. Fix the unreachable code or missing assignment.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below. It is a good practice to initialize a variable at declaration.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;               //Noncompliant
    /* Defect: val does not have a value if command is not 2 */
}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

#### Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that the initialization is not bypassed on some execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP33-C

# CERT C: Rule EXP34-C

Do not dereference null pointers

## Description

### Rule Definition

*Do not dereference null pointers.*

### Polyspace Implementation

The rule checker checks for **Null pointer**.

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Null pointer

#### Issue

**Null pointer** occurs when you use a pointer with a value of `NULL` as if it points to a valid memory location.

#### Risk

Dereferencing a null pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

#### Fix

Check a pointer for `NULL` before dereference.

If the issue occurs despite an earlier check for `NULL`, look for intermediate events between the check and the subsequent dereference. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

#### Example - Null pointer error

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
```

```
{
 int* p=NULL;

 *p=arr[0]; //Noncompliant
 /* Defect: Null pointer dereference */

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

The pointer `p` is initialized with value of `NULL`. However, when the value `arr[0]` is written to `*p`, `p` is assumed to point to a valid memory location.

**Correction — Assign Address to Null Pointer Before Dereference**

One possible correction is to initialize `p` with a valid memory address before dereference.

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 /* Fix: Assign address to null pointer */
 int* p=&arr[0];

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP34-C

# CERT C: Rule EXP35-C

Do not modify objects with temporary lifetime

## Description

### Rule Definition

*Do not modify objects with temporary lifetime.*

### Polyspace Implementation

The rule checker checks for **Accessing object with temporary lifetime**.

## Examples

### Accessing object with temporary lifetime

#### Issue

**Accessing object with temporary lifetime** occurs when you attempt to read from or write to an object with temporary lifetime that is returned by a function call. In a structure or union returned by a function, and containing an array, the array members are temporary objects. The lifetime of temporary objects ends:

- When the full expression or full declarator containing the call ends, as defined in the C11 Standard.
- After the next sequence point, as defined in the C90 and C99 Standards. A sequence point is a point in the execution of a program where all previous evaluations are complete and no subsequent evaluation has started yet.

For C++ code, **Accessing object with temporary lifetime** raises a defect only when you write to an object with a temporary lifetime.

If the temporary lifetime object is returned by address, no defect is raised.

#### Risk

Modifying objects with temporary lifetime is undefined behavior and can cause abnormal program termination and portability issues.

#### Fix

Assign the object returned from the function call to a local variable. The content of the temporary lifetime object is copied to the variable. You can now modify it safely.

#### Example - Modifying Temporary Lifetime Object Returned by Function Call

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6
```

```
struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

/* func_temp() returns a struct value containing
* an array with a temporary lifetime.
*/
int func(void) {

/*Writing to temporary lifetime object is
 undefined behavior
 */
    return ++(func_temp().a[0]);  //Noncompliant
}

void main(void) {
    (void)func();
}
```

In this example, `func_temp()` returns by value a structure with an array member `a`. This member has temporary lifetime. Incrementing it is undefined behavior.

**Correction — Assign Returned Value to Local Variable Before Writing**

One possible correction is to assign the return of the call to `func_temp()` to a local variable. The content of the temporary object `a` is copied to the variable, which you can safely increment.

```
 #include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

int func(void) {

/* Assign object returned by function call to
 *local variable
 */
    struct S_Array s = func_temp();

/* Local variable can safely be
 *incremented
 */
    ++(s.a[0]);
    return s.a[0];
}

void main(void) {
    (void)func();
}
```

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP35-C

# CERT C: Rule EXP36-C

Do not cast pointers into more strictly aligned pointer types

## Description

### Rule Definition

*Do not cast pointers into more strictly aligned pointer types.*

### Polyspace Implementation

This checker checks for **Source buffer misaligned with destination buffer**

## Examples

### Source buffer misaligned with destination buffer

#### Issue

The issue **Source buffer misaligned with destination buffer** occurs when the source pointer and the target pointer in a pointer-to-pointer conversion do not align. The misalignment occurs for reasons such as:

- The source pointer points to a buffer that is smaller than what the target pointer points to.
- The source pointer points to a buffer that is larger than what the target pointer points to, but the buffer size is not an exact multiple of the destination buffer size.
- The source pointer is a `void *` pointer and the target pointer is not. Polyspace does not report violations on casts and implicit conversion from `NULL` and `(void*)0`.

Polyspace does not report violations of this rule if:

- You use an environment that allows conversion between misaligned pointers. For instance, in 32bit and 64bit `x86` environment, converting `char*` into `uint16_t*` does not terminate a program abnormally.
- You use a source pointer that aligns correctly to the target type. For instance:
  - You use the `alignas` specifier to match the alignment of the source pointer with that of the target pointer.
  - You use pointers returned by functions such as `aligned_alloc()`, `malloc()`, and `realloc()`. These pointers matches the alignment of the target pointer.

#### Risk

If the alignment of a pointer changes in a pointer-to-pointer conversion, dereferencing the converted pointer causes abnormal program termination.

#### Fix

Avoid changing the alignment of a pointer in a pointer-to-pointer conversion.

**Example — Change in Pointer Alignment During Conversion**

```
#include <string.h>
struct record {
  int len;
  /* ... */
};

int copyBuffer (char *data, int offset)
{
  struct record *tmp;
  struct record dest;
  tmp = (struct record *) (data + offset);   //Noncompliant
  memcpy (&dest, tmp, sizeof (dest));
  /* ... */

  return dest.len;
}
```

In this example, the function copyBuffer converts a char* pointer into a struct record* pointer, followed by a memcpy operation. The memcpy operation assumes a struct record* alignment of tmp, which leads to undefined behavior. The rule checker reports a violation of this rule.

To avoid the issue, use data + offset as the source argument of the memcpy operation instead of using an intermediate struct record* pointer.

**Example — Conversion from void* to int***

```
int *lookup (void *loc)
{
  /* ... */
  return loc; //Noncompliant
}

void search (char *pos)
{
  int *found = lookup (pos);
}
```

In this example, the lookup function has a void* parameter, but converts the parameter into an int* pointer when returning. In the function search(), calling the function lookup() with a char* pointer results in a conversion between a char* pointer and an int* pointer. Polyspace reports a violation of this rule.

**Example — Use alignas Specifier to Avoid Alignment Mismatch**

This example aligns the char pointer char_p as an int by using the alignas specifier. Because char_p is aligned as an int, converting it to an int* does not violate this rule.

```
#include <stdalign.h>
#include <assert.h>

void foo(void) {
  /* Align char_p to the alignment of an int */
  alignas(int) char char_p = 'a';
  int *int_p = (int *)&char_p; //Compliant
  char *char_new = (char *)int_p;
  /* Both char_new and &char_p point to equally aligned objects */
}
```

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

EXP36-C

# CERT C: Rule EXP37-C

Call functions with the correct number and type of arguments

## Description

### Rule Definition

*Call functions with the correct number and type of arguments.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Bad file access mode or status**.
- **Unreliable cast of function pointer**.
- **Standard function call with incorrect arguments**.
- **Unsupported complex arguments**
- **Function declaration mismatch**
- **Incompatible argument**

## Examples

### Bad file access mode or status

#### Issue

**Bad file access mode or status** occurs when you use functions in the `fopen` or `open` group with invalid or incompatible file access modes, file creation flags, or file status flags as arguments. For instance, for the `open` function, examples of valid:

- Access modes include `O_RDONLY`, `O_WRONLY`, and `O_RDWR`
- File creation flags include `O_CREAT`, `O_EXCL`, `O_NOCTTY`, and `O_TRUNC`.
- File status flags include `O_APPEND`, `O_ASYNC`, `O_CLOEXEC`, `O_DIRECT`, `O_DIRECTORY`, `O_LARGEFILE`, `O_NOATIME`, `O_NOFOLLOW`, `O_NONBLOCK`, `O_NDELAY`, `O_SHLOCK`, `O_EXLOCK`, `O_FSYNC`, `O_SYNC` and so on.

The defect can occur in the following situations.

| Situation | Risk | Fix |
|---|---|---|
| You pass an empty or invalid access mode to the `fopen` function.<br>According to the ANSI C standard, the valid access modes for `fopen` are:<br>• `r`,`r+`<br>• `w`,`w+`<br>• `a`,`a+`<br>• `rb`, `wb`, `ab`<br>• `r+b`, `w+b`, `a+b`<br>• `rb+`, `wb+`, `ab+` | `fopen` has undefined behavior for invalid access modes.<br>Some implementations allow extension of the access mode such as:<br>• GNU: `rb+cmxe,ccs=utf`<br>• Visual C++: `a+t`, where `t` specifies a text mode.<br>However, your access mode string must begin with one of the valid sequences. | Pass a valid access mode to `fopen`. |
| You pass the status flag `O_APPEND` to the `open` function without combining it with either `O_WRONLY` or `O_RDWR`. | `O_APPEND` indicates that you intend to add new content at the end of a file. However, without `O_WRONLY` or `O_RDWR`, you cannot write to the file.<br>The `open` function does not return -1 for this logical error. | Pass either `O_APPEND\|O_WRONLY` or `O_APPEND\|O_RDWR` as access mode. |
| You pass the status flags `O_APPEND` and `O_TRUNC` together to the `open` function. | `O_APPEND` indicates that you intend to add new content at the end of a file. However, `O_TRUNC` indicates that you intend to truncate the file to zero. Therefore, the two modes cannot operate together.<br>The `open` function does not return -1 for this logical error. | Depending on what you intend to do, pass one of the two modes. |
| You pass the status flag `O_ASYNC` to the `open` function. | On certain implementations, the mode `O_ASYNC` does not enable signal-driven I/O operations. | Use the `fcntl(pathname, F_SETFL, O_ASYNC);` instead. |

**Fix**

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.

- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Invalid Access Mode with fopen

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "rw"); //Noncompliant
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

In this example, the access mode `rw` is invalid. Because `r` indicates that you open the file for reading and `w` indicates that you create a new file for writing, the two access modes are incompatible.

#### Correction — Use Either r or w as Access Mode

One possible correction is to use the access mode corresponding to what you intend to do.

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "w");
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

### Unreliable cast of function pointer

#### Issue

**Unreliable cast of function pointer** occurs when a function pointer is cast to another function pointer that has different argument or return type.

This defect applies only if the code language for the project is C.

#### Risk

If you cast a function pointer to another function pointer with different argument or return type and then use the latter function pointer to call a function, the behavior is undefined.

#### Fix

Avoid a cast between two function pointers with mismatch in argument or return types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.

- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example - Unreliable cast of function pointer error**

```
#include <stdio.h>
#include <math.h>
#include <stdio.h>
#define PI 3.142

double Calculate_Sum(int (*fptr)(double))
{
    double  sum = 0.0;
    double  y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;

    fp = sin;
    sum = Calculate_Sum(fp);  //Noncompliant
    /* Defect: fp implicitly cast to int(*) (double) */

    printf("sum(sin): %f\n", sum);
    return 0;
}
```

The function pointer `fp` is declared as `double (*)(double)`. However in passing it to function `Calculate_Sum`, `fp` is implicitly cast to `int (*)(double)`.

**Correction — Avoid Function Pointer Cast**

One possible correction is to check that the function pointer in the definition of `Calculate_Sum` has the same argument and return type as `fp`. This step makes sure that `fp` is not implicitly cast to a different argument or return type.

```
#include <stdio.h>
#include <math.h>
#include <stdio.h>
# define PI 3.142

/*Fix: fptr has same argument and return type everywhere*/
double Calculate_Sum(double (*fptr)(double))
{
    double  sum = 0.0;
    double y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;


    fp = sin;
    sum = Calculate_Sum(fp);
    printf("sum(sin): %f\n", sum);

    return 0;
}
```

### Standard function call with incorrect arguments

#### Issue

**Standard function call with incorrect arguments** occurs when the arguments to certain standard functions do not meet the requirements for their use in the functions.

For instance, the arguments to these functions can be invalid in the following ways.

| Function Type | Situation | Risk | Fix |
|---|---|---|---|
| String manipulation functions such as `strlen` and `strcpy` | The pointer arguments do not point to a `NULL`-terminated string. | The behavior of the function is undefined. | Pass a `NULL`-terminated string to string manipulation functions. |
| File handling functions in `stdio.h` such as `fputc` and `fread` | The `FILE*` pointer argument can have the value `NULL`. | The behavior of the function is undefined. | Test the `FILE*` pointer for `NULL` before using it as function argument. |
| File handling functions in `unistd.h` such as `lseek` and `read` | The file descriptor argument can be -1. | The behavior of the function is undefined.<br>Most implementations of the `open` function return a file descriptor value of -1. In addition, they set `errno` to indicate that an error has occurred when opening a file. | Test the return value of the `open` function for -1 before using it as argument for `read` or `lseek`.<br>If the return value is -1, check the value of `errno` to see which error has occurred. |

| Function Type | Situation | Risk | Fix |
|---|---|---|---|
| | The file descriptor argument represents a closed file descriptor. | The behavior of the function is undefined. | Close the file descriptor only after you have completely finished using it. Alternatively, reopen the file descriptor before using it as function argument. |
| Directory name generation functions such as `mkdtemp` and `mkstemps` | The last six characters of the string template are not XXXXXX. | The function replaces the last six characters with a string that makes the file name unique. If the last six characters are not XXXXXX, the function cannot generate a unique enough directory name. | Test if the last six characters of a string are XXXXXX before using the string as function argument. |
| Functions related to environment variables such as `getenv` and `setenv` | The string argument is `""`. | The behavior is implementation-defined. | Test the string argument for `""` before using it as `getenv` or `setenv` argument. |
| | The string argument terminates with an equal sign, =. For instance, `"C="` instead of `"C"`. | The behavior is implementation-defined. | Do not terminate the string argument with =. |
| String handling functions such as `strtok` and `strstr` | • `strtok`: The delimiter argument is `""`.<br>• `strstr`: The search string argument is `""`. | Some implementations do not handle these edge cases. | Test the string for `""` before using it as function argument. |

**Fix**

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - NULL Pointer Passed as strnlen Argument

```
#include <string.h>
#include <stdlib.h>

enum {
    SIZE10 = 10,
    SIZE20 = 20
};

int func() {
    char* s = NULL;
    return strnlen(s, SIZE20); //Noncompliant
}
```

In this example, a NULL pointer is passed as strnlen argument instead of a NULL-terminated string.

Before running analysis on the code, specify a GNU compiler. See Compiler (-compiler).

#### Correction — Pass NULL-terminated String

Pass a NULL-terminated string as the first argument of strnlen.

```
#include <string.h>
#include <stdlib.h>

enum {
    SIZE10 = 10,
    SIZE20 = 20
};

int func() {
    char* s = "";
    return strnlen(s, SIZE20);
}
```

### Unsupported complex arguments

#### Issue

**Unsupported complex arguments** occurs when these functions are called with a complex argument:

- atan2
- erf
- fdim
- fmin
- ilogb
- llround
- logb
- nextafter
- rint
- tgamma

- `cbrt`
- `erfc`
- `floor`
- `fmod`
- `ldexp`
- `log10`
- `lrint`
- `nexttoward`
- `round`
- `trunc`
- `ceil`
- `exp2`
- `fma`
- `frexp`
- `lgamma`
- `log1p`
- `round`
- `remainder`
- `scalbn`
- `copysign`
- `expm1`
- `fmax`
- `hypot`
- `llrint`
- `log2`
- `nearbyint`
- `remquo`
- `scalbln`

**Risk**

Calling any of the preceding functions with a `complex` argument is undefined behavior in the C++ standard, which might lead to unexpected results. Because some mathematical functions support `complex` arguments while the functions in the preceding list do not, the unexpected results might be difficult to debug. Performing some of these mathematical operations on a `complex` number might not be mathematically sound, which indicates an issue in the underlying logic of your code.

**Fix**

Avoid calling the preceding functions with a `complex` input argument. To perform the preceding mathematical operations on a complex number, define alternative functions that support `complex` arguments.

**Example — Calling `log2` and `trunc` Functions with Complex Arguments**

```
#include <complex.h>
#include <tgmath.h>
```

```
typedef double complex cDouble;
cDouble Noncompliant (void)
{
    cDouble Z = 2.0 + 4.0 * I;
    cDouble result = log2 (Z); //Noncompliant
    return trunc(result);//Noncompliant
}
```

In this example, the function `Noncompliant` calculates the base two logarithm of a complex number, truncates the result, and returns it. The functions `log2` and `trunc` do not support a complex argument. Polyspace flags these operations. To run this example, specify `gnu6.x` as the compiler. For instance, in the command line, use the option `-compiler gnu6.x`.

#### Correction — Define Functions That Support Complex Arguments

One possible correction is to define alternative functions that support complex numbers. For instance, while `log2` does not support `complex` numbers, the function `log` does. Define a function `complexLog2` that uses `log` to calculate the base two logarithm of a complex number. Similarly, `trunc` does not support `complex` numbers and the mathematical rule for truncating a complex number is not well-defined. Define a function `complexTrunc` that truncates a complex number by truncating its real and imaginary parts separately.

```
#include <complex.h>
#include <tgmath.h>
typedef double complex cDouble;

cDouble complexLog2(cDouble z) {
    return log (z) / log (2);  // Compliant
}

cDouble complexTrunc(cDouble z){
    return trunc(creal(z)) + I*trunc(cimag(z));  //Compliant
}

cDouble Compliant (void)
{
    cDouble Z = 2.0 + 4.0 * I;
    cDouble result = complexLog2 (Z); //Compliant
    return complexTrunc(result);//Compliant
}
```

### Function declaration mismatch

#### Issue

**Function declaration mismatch** occurs when the prototype of a function does not match its definition. If a function lacks a prototype in the file where it is called, Polyspace deduces its prototype based on the signature of the call. If the deduced prototype does not match the definition of the function, Polyspace raises this defect. The prototype of a variadic function cannot be deduced from its function call. If you call a variadic function without specifying its prototype in the same file, Polyspace raises this defect.

When deducing the prototype of a function from a call to such a function, Polyspace makes these assumptions:

- The number of arguments of the deduced prototype is equal to the input argument of the function call.

- The argument types of the deduced prototype are set by implicitly promoting the argument types of the function call. For instance, both signed and unsigned `char` or `short` type arguments are promoted to `int`. Similarly `float` type arguments are promoted to `double`.
- Type mismatch between the arguments of the function definition and the function prototype might depend on your environment. Polyspace considers two types as compatible if they have the same size and signedness in the environment that you use. For instance, if your specify `-target` as i386, Polyspace considers `long` and `int` as compatible types.

*The checker does not flag this issue in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

**Risk**

According to the C standard, function declaration mismatch might result in undefined behavior even though such code might compile successfully producing only warnings during compilation. Because code with this issue might compile successfully, function declaration mismatches might result in unexpected results that are difficult to diagnose.

**Fix**

- Before you call a function, provide its complete prototype, even if you define the function later in the same file.
- Avoid any mismatch between the number arguments in the function prototype declaration and the function definition.
- Avoid any mismatch between the argument types of the function prototype declaration and the function definition.

When complete prototypes of the called functions are provided, the compiler tries to resolve any function declaration mismatches through implicit casting. If the compiler fails to resolve the mismatch, the compilation fails, which prevents unexpected behavior. To fix such compile errors, call the functions using argument types and numbers that match the function definition.

**Example — Function Calls That Lack Prototypes**

```
// file1.c
void foo(int iVar){
    //...
}
void bar(float fVar1, float fVar2){
    //...
}
void bar2(float fVar1){
    //...
}
void fubar(const char* str,...){
    //...
}
void foo2(char cVar){
    //...
}
void call_variadic(){
    fubar("String");
}
```

```
//file2.c
void bar2(float);
void foo2(int);
void call_funcs(){
    int iTemp;
    float fTemp;
    foo();//Noncompliant
    bar(fTemp,fTemp);//Noncompliant
    fubar("String"); //Noncompliant
    bar2(iTemp);//Compliant
    foo2(iTemp); //Noncompliant
}
```

In this example, the functions foo, foo2, bar, bar2, and fubar are defined in the file file1.c. These functions are then called in the file file2.c.

- The function foo is defined in file1.c with one int input and called in file2.c without any input. Because file2.c does not have a prototype for foo, Polyspace deduces a prototype based on the call foo(), which takes no input. This deduced prototype does not match the function declaration in file1.c. Polyspace flags the call.
- The function bar is defined in file1.c with two float inputs and called in file2.c with two float inputs. Because file2.c does not have a prototype for bar, Polyspace deduces a prototype based on the call bar(fTemp,fTemp). By promoting the argument types of the function call, the signature of this deduced prototype is bar(double, double), which does not match the function declaration in file1.c. Polyspace flags the call.
- The function bar2 is defined in file1.c with one float input. The complete prototype for bar2, which matches the definition, is provided in file2.c. Because a complete prototype is present in this file, when bar2 is called with an incorrect input, the compiler implicitly converts the int input iTemp into a float. Because the call to the function matches the declaration after an implicit conversion facilitated by the prototype, Polyspace does not flag the call.
- The function foo2 is defined in file1.c with a char input. Its prototype in file2.c is defined with a int input. Because the definition and the prototype do not match, Polyspace flags the call to foo2.
- The variadic function fubar is defined in file1.c. The call to it in call_variadic is compliant because the call comes after the definition. The function fubar does not have a prototype in file2.c. Because the function takes a variable number of inputs, its prototype cannot be deduced. The call to fubar in file2.c lacks a prototype and Polyspace flags the call.

#### Correction — Compliant Function Calls

The fix for this defect is to declare complete prototypes for the called functions in all compilation modules. It is a best practice to combine the function prototype declarations in a header file, and then include it in files where the functions are called. In this case, resolve the flagged issues by including such a header file prototype.h in file2.c. Once a correct prototype is declared, the call foo() in file2.c causes a compilation failure because the compiler cannot resolve the mismatch between the call and the declared prototype. Call foo with an int to resolve the compilation failure.

```
// file1.c
void foo(int iVar){
    //...
}
void bar(float fVar1, float fVar2){
    //...
}
void bar2(float fVar1){
    //...
}
void fubar(const char* str,...){
    //...
}
void foo2(char cVar){
    //...
}
void call_variadic(){
    fubar("String");
}
```

```
//prototypes.h
void foo(int iVar);
void bar(float fVar1, float fVar2);
void fubar(const char* str,...);
void bar2(float);
void foo2(char);
void call_variadic(void);
void call_funcs(void);
```

```
//file2.c
#include"prototype.h"
void call_funcs(){
    int iTemp;
    float fTemp;
    //foo(); This call results in compile failure
    foo(iTemp);//Compliant
    bar(fTemp,fTemp);//Compliant
    fubar("String"); //Compliant
    bar2(iTemp);//Compliant
    foo2('a'); //Compliant
}
```

### Incompatible argument

#### Issue

**Incompatible argument** occurs when an external function is called by using an argument that is not compatible with the prototype. The compatibility of types might depend on the set of hardware and software that you use. For instance, consider this code:

```
extern long foo(int);

long bar(long i) {
    return foo(i); //Noncompliant: calls foo(int) with a long
}
```

The external function `foo` is called with a `long` when an `int` is expected. In environments where the size of an `int` is smaller than the size of a `long`, this function call is incompatible with the prototype, resulting in a defect.

#### Risk

Calling external functions with arguments that are incompatible with the parameter is undefined behavior. Depending on your environment, the code might compile but behave in an unexpected way.

#### Fix

When calling external functions, use argument types that are smaller or equal in size compared to the parameter type defined in the prototype. Check the sizes of various integer types in your environment to determine compatibility of argument and parameter types.

#### Example — Call External Functions with Incompatible argument

```
extern long foo1(int);
extern long foo2(long);
void bar(){
    int varI;
    long varL;
    foo1(varL);//Noncompliant
    foo2(varI);//Compliant
}
```

In this example, the external function `foo1` is called with a `long` argument, while the prototype specifies the parameter as an `int`. In `x86` architecture, the size of `long` is larger than the size of `int`. The call `foo1(varL)` might result in undefined behavior. Polyspace flags the call. The call `foo2(varI)` uses an `int` argument while the parameter is specified as a `long`. This type of mismatch is compliant with this rule because the size of `int` is not larger than the size of `long`.

To run this example in Polyspace, use these options:

- `-lang c`
- `-target x86_64`

See `Target processor type (-target)`.

#### Correction — Cast Variables Explicitly to Match Argument to Parameter

To fix this issue, cast the argument of `foo1` explicitly so that argument type and parameter type matches.

```
extern long foo1(int);
extern long foo2(long);
void bar(){
    int varI;
    long varL;
    foo1((int)varL);//Compliant
    foo2(varI);//Compliant
}
```

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP37-C

# CERT C: Rule EXP39-C

Do not access a variable through a pointer of an incompatible type

## Description

### Rule Definition

*Do not access a variable through a pointer of an incompatible type.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Cast to pointer pointing to object of different type**
- **Reading memory reallocated from object of another type without reinitializing first**

.

## Examples

### Cast to pointer pointing to object of different type

#### Issue

The issue occurs when you perform a cast between a pointer to an object type and a pointer to a different object type.

#### Risk

If a pointer to an object is cast into a pointer to a different object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.

Even if the conversion produces a pointer that is correctly aligned, the behavior can be undefined if the pointer is used to access an object.

Exception: You can convert a pointer to object type into a pointer to one of the following types:

- `char`
- `signed char`
- `unsigned char`

#### Example - Noncompliant: Cast to Pointer Pointing to Object of Wider Type

```
signed   char *p1;
unsigned int *p2;

void foo(void){
  p2 = ( unsigned int * ) p1;     /* Non-compliant */
}
```

In this example, `p1` can point to a `signed char` object. However, `p1` is cast to a pointer that points to an object of wider type, `unsigned int`.

#### Example - Noncompliant: Cast to Pointer Pointing to Object of Narrower Type

```
extern unsigned int read_value ( void );
extern void display ( unsigned int n );

void foo ( void ){
  unsigned int u = read_value ( );
  unsigned short *hi_p = ( unsigned short * ) &u;    /* Non-compliant  */
  *hi_p = 0;
  display ( u );
}
```

In this example, u is an `unsigned int` variable. &u is cast to a pointer that points to an object of narrower type, `unsigned short`.

On a big-endian machine, the statement `*hi_p = 0` attempts to clear the high bits of the memory location that &u points to. But, from the result of `display(u)`, you might find that the high bits have not been cleared.

#### Example - Compliant: Cast Adding a Type Qualifier

```
const short *p;
const volatile short *q;
void foo (void){
  q = ( const volatile short * ) p;  /* Compliant */
}
```

In this example, both p and q can point to `short` objects. The cast between them adds a `volatile` qualifier only and is therefore compliant.

### Reading memory reallocated from object of another type without reinitializing first

#### Issue

This issue occurs when you do the following in sequence:

1. Reallocate memory to an object with a type that is different from the original allocation.

   For instance, in this code snippet, a memory originally allocated to a pointer with type `struct A*` is reallocated to a pointer with type `struct B*`:

   ```
   struct A;
   struct B;

   struct A *Aptr = (struct A*) malloc(sizeof(struct A));
   struct B *Bptr = (struct B*) realloc(Aptr, sizeof(struct B));
   ```
2. Read from this reallocated memory without reinitializing the memory first.

   Read accesses on the pointer to the reallocated memory can happen through pointer dereference or array indexing. Passing the pointer to a function that takes a pointer to a `const`-qualified object as the corresponding parameter also counts as a read access.

#### Risk

Reading from reallocated memory that has not been reinitialized leads to undefined behavior.

#### Fix

Reinitialize memory after reallocation and before the first read access.

The checker considers any write access on the pointer to the reallocated memory as satisfying the reinitialization requirement (even if the object might only be partially reinitialized). Write accesses on the pointer to the reallocated memory can happen through pointer dereference or array indexing. Passing the pointer to a function that takes a pointer to a non-`const`-qualified object as the corresponding parameter also counts as a write access.

**Example – Noncompliant: Reading from Reallocated Memory Without Reinitializing First**

```
#include<stdlib.h>

struct group {
    char *groupFirst;
    int groupSize;
};

struct groupWithID {
    int groupID;
    char *groupFirst;
    int groupSize;
};

char* readName();
int readSize();

void createGroup(int nextAvailableID) {
    struct group *aGroup;
    struct groupWithID *aGroupWithID;

    aGroup = (struct group*) malloc(sizeof(struct group));

    if(!aGroup) {
        /*Handle error*/
    }

    aGroup->groupFirst = readName();
    aGroup->groupSize  = readSize();

    /* Reassign to group with ID */
    aGroupWithID = (struct groupWithID*) realloc(aGroup, sizeof(struct groupWithID));
    if(!aGroupWithID) {
        free(aGroup);
        /*Handle error*/
    }

    if(aGroupWithID -> groupSize > 0) { /* Noncompliant */
        /* ... */
    }

    /* ...*/
    free(aGroupWithID);
}
```

In this example, the memory allocated to a `group*` pointer using the `malloc` function is reallocated to a `groupWithID*` pointer using the `realloc` function. There is a read access on the reallocated memory before the memory is reinitialized.

#### Correction – Reinitialize Memory After Reallocation and Before First Read

Reinitialize the memory assigned to the `groupWithID*` pointer before the first read access. All bits of the memory can be reinitialized using the `memset` function.

```
#include<stdlib.h>
#include<string.h>

struct group {
    char *groupFirst;
    int groupSize;
};

struct groupWithID {
    int groupID;
    char *groupFirst;
    int groupSize;
};

char* readName();
int readSize();

void createGroup(int nextAvailableID) {
    struct group *aGroup;
    struct groupWithID *aGroupWithID;

    aGroup = (struct group*) malloc(sizeof(struct group));

    if(!aGroup) {
        /*Handle error*/
    }

    aGroup->groupFirst = readName();
    aGroup->groupSize  = readSize();

    /* Reassign to group with ID */
    aGroupWithID = (struct groupWithID*) realloc(aGroup, sizeof(struct groupWithID));
    if(!aGroupWithID) {
        free(aGroup);
        /*Handle error*/
    }

    memset(aGroupWithID, 0 , sizeof(struct groupWithID));
    /* Reinitialize group */
    if(aGroupWithID -> groupSize > 0) {
        /* ... */
    }

    /* ...*/
    free(aGroupWithID);
}
```

## Check Information

**Group:** Rule 03. Expressions (EXP)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP39-C

# CERT C: Rule EXP40-C

Do not modify constant objects

## Description

### Rule Definition

*Do not modify constant objects.*

### Polyspace Implementation

The rule checker checks for **Writing to const qualified object**.

## Examples

### Writing to const qualified object

#### Issue

**Writing to `const` qualified object** occurs when you do one of the following:

- Use a `const`-qualified object as the destination of an assignment.
- Pass a `const`-qualified object to a function that modifies the argument.

For instance, the defect can occur in the following situations:

- You pass a `const`-qualified object as first argument of one of the following functions:
  - `mkstemp`
  - `mkostemp`
  - `mkostemps`
  - `mkdtemp`
- You pass a `const`-qualified object as the destination argument of one of the following functions:
  - `strcpy`
  - `strncpy`
  - `strcat`
  - `memset`
- You perform a write operation on a `const`-qualified object.

#### Risk

The risk depends upon the modifications made to the `const`-qualified object.

| Situation | Risk |
|---|---|
| Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on. | These functions replace the last six characters of their first argument with a string. Therefore, they expect a modifiable `char` array as their first argument. |
| Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on. | These functions modify their destination argument. Therefore, they expect a modifiable `char` array as their destination argument. |
| Writing to the object | The `const` qualifier implies an agreement that the value of the object will not be modified. By writing to a `const`-qualified object, you break the agreement. The result of the operation is undefined. |

#### Fix

The fix depends on the modification made to the `const`-qualified object.

| Situation | Fix |
|---|---|
| Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on. | Pass a non-`const` object as first argument of the function. |
| Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on. | Pass a non-`const` object as destination argument of the function. |
| Writing to the object | Perform the write operation on a non-`const` object. |

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Writing to `const`-Qualified Object

```
#include <string.h>

const char* buffer = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string); //Noncompliant
}
```

In this example, because `buffer` is `const`-qualified, `strchr(buffer,'X')` returns a `const`-qualified `char*` pointer. When this `char*` pointer is used as the destination argument of `strcpy`, a **Writing to const qualified object** error appears.

#### Correction — Copy const-Qualified Object to Non-const Object

One possible correction is to assign the constant string to a non-const object and use the non-const object as destination argument of strchr.

```
#include <string.h>

char buffer[] = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C (-cert-c)

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP40-C

# CERT C: Rule EXP42-C

Do not compare padding data

## Description

### Rule Definition

*Do not compare padding data.*

### Polyspace Implementation

The rule checker checks for **Memory comparison of padding data**.

## Examples

### Memory comparison of padding data

#### Issue

**Memory comparison of padding data** occurs when you use the memcmp function to compare two structures as a whole. In the process, you compare meaningless data stored in the structure padding.

For instance:

```
typedef struct structType {
    char member1;
    int member2;
    //...
}myStruct;

myStruct var1;
myStruct var2;
//...
if(memcmp(&var1,&var2,sizeof(var1)))
{//...}
```

#### Risk

If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. For an example of padding, see Higher Estimate of Size of Local Variables.

The content of these extra padding bytes is meaningless. The C Standard allows the content of these bytes to be indeterminate, giving different compilers latitude to implement their own padding. If you perform a byte-by-byte comparison of structures with memcmp, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.

#### Fix

Instead of comparing two structures in one attempt, compare the structures member by member.

For efficient code, write a function that does the comparison member by member. Use this function for comparing two structures.

You can use `memcmp` for byte-by-byte comparison of structures only if you know that the structures do not contain padding. Typically, to prevent padding, you use specific attributes or pragmas such as `#pragma pack`. However, these attributes or pragmas are not supported by all compilers and make your code implementation-dependent. If your structures contain bit-fields, using these attributes or pragmas cannot prevent padding.

**Example - Structures Compared with `memcmp`**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{

    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    if (0 == memcmp(left, right, sizeof(S_Padding)))  //Noncompliant
    {
        return 1;
    }
    else
        return 0;
}
```

In this example, `memcmp` compares byte-by-byte the two structures that `left` and `right` point to. Even if the values stored in the structure members are the same, the comparison can show an inequality if the meaningless values in the padding bytes are not the same.

**Correction — Compare Structures Member by Member**

One possible correction is to compare individual structure members.

**Note** You can compare entire arrays by using `memcmp`. All members of an array have the same data type. Padding bytes are not required to store arrays.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{
    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    return ((left->c == right->c) &&
            (left->i == right->i) &&
            (left->bf1 == right->bf1) &&
            (left->bf2 == right->bf2) &&
            (memcmp(left->buffer, right->buffer, 20) == 0));
}
```

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C (-cert-c)

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

EXP42-C

# CERT C: Rule EXP43-C

Avoid undefined behavior when using restrict-qualified pointers

## Description

### Rule Definition

*Avoid undefined behavior when using restrict-qualified pointers.*

### Polyspace Implementation

The rule checker checks for **Overlapping access by `restrict`-qualified pointers**.

## Examples

### Overlapping Access by `restrict`-Qualified Pointers

#### Issue

**Overlapping access by `restrict`-qualified pointers** occurs when any of these is true:

- Two `restrict`-qualified pointers modify objects that have overlapping memory addresses.
- In a function, one or more `const restrict`-qualified pointer might modify a non`const restrict`-qualified pointer but when calling the function, the `const` and `nonconst` arguments are the same or are derived from the same pointer.
- A standard library function is called by using restrict-qualified pointers that overlap.
- A `restrict`-qualified pointer is assigned to another `restrict`-qualified pointer within the same scope.

#### Risk

The `restrict` qualifier on a pointer implies that within a block, only this pointer (or other pointers created from this pointer) can access the pointed object. You cannot create a second pointer independently of the `restrict`-qualified pointer to point to the object.

This specification requires that two `restrict`-qualified pointers cannot point to the same or overlapping objects. Two `restrict`-qualified pointers accessing the same memory address or object results in undefined behavior.

#### Fix

If you assign a `restrict`-qualified pointer to another pointer, make sure that the destination pointer itself is not `restrict`-qualified. The `restrict` qualifier assists the compiler in optimizing the code. Removing all instances of this qualifier does not change the observable code behavior.

#### Example — Assignment Between `restrict`-qualified Pointers in the Same Scope

```
int *restrict rptr1;
int *restrict rptr2;
int *         ptr;
extern int arr[];
```

```
void func (void)
{
  arr[0] = 0;
  arr[1] = 1;
  rptr1 = &arr[0];
  rptr2 = &arr[1];
  rptr2 = rptr1;  //Non-compliant
  ptr   = rptr1;  //Compliant
  /* ... */

}
```

In this example, `rptr1` and `rptr2` are `restrict`-qualified pointers that point to different locations in the same array `arr`. Assigning one such `restrict`-qualified pointer to another in the same scope causes a rule violation.

**Example — `restrict`-Qualified Function Parameters**

```
#include <stddef.h>
#include <stdio.h>
void addArray (size_t n, int *restrict res,
               const int *restrict lhs, const int *restrict rhs)
{
  for (size_t i = 0; i < n; ++i) {
    res[i] = lhs[i] + rhs[i];
  }
}
void foo (void)
{
  int a[100];
  memset(&a, 0, 100);
  addArray (100, a, a, a);//Noncompliant
}
```

In this example, the function `addArray()` is defined with three `restrict` qualified parameters `lhs`, `rhs`, and `res`. The parameters `lhs` and `rhs` are `const restrict` pointers. The function `addArray()` is then invoked by using the same pointer `a` as all three parameters. As a result, the `const restrict` qualified pointers `lhs` and `rhs` might attempt to access the memory associated with the non-`const restrict` qualified pointer `res`. This overlapping access between `lhs` and `res` as well as that between `rhs` and `res` are undefined behaviors. Polyspace raises two violations on the function call.

**Example — Invoking Library Functions by Using `restrict`-Qualified Pointers**

```
#include <string.h>

void func(void) {
  char c_str[]= "test string";
  char *ptr1 = c_str;
  char *ptr2;

  ptr2 = ptr1 + 3;
  /* Undefined behavior because of overlapping objects */
  memcpy(ptr2, ptr1, 6);//Noncompliant
  /* ... */
}
```

In this example, the function `memcpy` has two `restrict`-qualified pointers as input parameters This function copies six bytes from the location pointed to by `ptr1` into the location pointed to by `ptr2`.

Because the distance between `ptr1` and `ptr2` is three bytes, the `memcpy` function call results in two `restrict`-qualified pointers attempting to modify overlapping memory. This overlapping access results in undefined behavior. Polyspace flags the call to `memcpy`.

**Example — Assignments Between Restricted Pointers**

```
void func(void) {
  int *restrict p1;
  int *restrict p2 = p1; /* Undefined behavior */ //Noncompliant
 }
```

In this example, the `restrict`-qualified pointer `p1` is assigned to another `restrict`-qualified pointer in the same scope. Such assignments result in undefined behavior, which Polyspace flags. To resolve the issue, declare `p2` in a separate nested scope. For example:

```
void func(void) {
  int *restrict p1;
  {
      int *restrict p2 = p1;//Compliant
  }
 }
```

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP43-C

# CERT C: Rule EXP44-C

Do not rely on side effects in operands to sizeof, _Alignof, or _Generic

## Description

### Rule Definition

*Do not rely on side effects in operands to sizeof, _Alignof, or _Generic.*

### Polyspace Implementation

The rule checker checks for **Side effect of expression ignored**.

## Examples

### Side effect of expression ignored

#### Issue

**Side effect of expression ignored** occurs when the `sizeof`, `_Alignof`, or `_Generic` operator operates on an expression with a side effect. When evaluated, an expression with side effect modifies at least one of the variables in the expression.

For instance, the defect checker does not flag `sizeof(n+1)` because `n+1` does not modify `n`. The checker flags `sizeof(n++)` because `n++` is intended to modify `n`.

The check also applies to the C++ operator `alignof` and its C extensions, `__alignof__` and `__typeof__`.

#### Risk

The expression in a `_Alignof` or `_Generic` operator is not evaluated. The expression in a `sizeof` operator is evaluated only if it is required for calculating the size of a variable-length array, for instance, `sizeof(a[n++])`.

When an expression with a side effect is not evaluated, the variable modification from the side effect does not happen. If you rely on the modification, you can see unexpected results.

#### Fix

Evaluate the expression with a side effect in a separate statement, and then use the result in a `sizeof`, `_Alignof`, or `_Generic` operator.

For instance, instead of:

```
a = sizeof(n++);
```

perform the operation in two steps:

```
n++;
a = sizeof(n);
```

The checker considers a function call as an expression with a side effect. Even if the function does not have side effects now, it might have side effects on later additions. The code is more maintainable if you call the function outside the `sizeof` operator.

#### Example - Increment Operator in `sizeof`

```
#include <stdio.h>

void func(void) {
    unsigned int a = 1U;
    unsigned int b = (unsigned int)sizeof(++a); //Noncompliant
    printf ("%u, %u\n", a, b);
}
```

In this example, `sizeof` operates on `++a`, which is intended to modify `a`. Because the expression is not evaluated, the modification does not happen. The `printf` statement shows that `a` still has the value `1`.

#### Correction — Perform Increment Outside `sizeof`

One possible correction is to perform the increment first, and then provide the result to the `sizeof` operator.

```
#include <stdio.h>

void func(void) {
    unsigned int a = 1U;
    ++a;
    unsigned int b = (unsigned int)sizeof (a);
    printf ("%u, %u\n", a, b);
}
```

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP44-C

# CERT C: Rule EXP45-C

Do not perform assignments in selection statements

## Description

### Rule Definition

*Do not perform assignments in selection statements.*

### Polyspace Implementation

The rule checker checks for **Invalid use of = (assignment) operator**.

## Examples

### Invalid use of = (assignment) operator

#### Issue

**Invalid use of = operator** occurs when an assignment is made inside the predicate of a conditional, such as `if` or `while`.

In C and C++, a single equal sign is an assignment not a comparison. Using a single equal sign in a conditional statement can indicate a typo or a mistake.

#### Risk

- Conditional statement tests the wrong values— The single equal sign operation assigns the value of the right operand to the left operand. Then, because this assignment is inside the predicate of a conditional, the program checks whether the new value of the left operand is nonzero or not NULL.
- Maintenance and readability issues — Even if the assignment is intended, someone reading or updating the code can misinterpret the assignment as an equality comparison instead of an assignment.

#### Fix

- If the assignment is a bug, to check for equality, add a second equal sign (==).
- If the assignment inside the conditional statement was intentional, to improve readability, separate the assignment and the test. Move the assignment outside the control statement. In the control statement, simply test the result of the assignment.

  If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

  - “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
  - “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
  - “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Example - Single Equal Sign Inside an `if` Condition

```
#include <stdio.h>

void bad_equals_ex(int alpha, int beta)
{
    if(alpha = beta) //Noncompliant
    {
        printf("Equal\n");
    }
}
```

The equal sign is flagged as a defect because the assignment operator is used within the predicate of the if-statement. The predicate assigns the value `beta` to `alpha`, then implicitly tests whether `alpha` is true or false.

### Correction — Change Expression to Comparison

One possible correction is adding an additional equal sign. This correction changes the assignment to a comparison. The if condition compares whether `alpha` and `beta` are equal.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha == beta)
    {
        printf("Equal\n");
    }
}
```

### Correction — Assignment and Comparison Inside the `if` Condition

If an assignment must be made inside the predicate, a possible correction is adding an explicit comparison. This correction assigns the value of `beta` to `alpha`, then explicitly checks whether `alpha` is nonzero. The code is clearer.

```
#include <stdio.h>

int assignment_not_zero(int alpha, int beta)
{
    if((alpha = beta) != 0)
    {
        return alpha;
    }
    else
    {
        return 0;
    }
}
```

### Correction — Move Assignment Outside the `if` Statement

If the assignment can be made outside the control statement, one possible correction is to separate the assignment and comparison. This correction assigns the value of `beta` to `alpha` before the if. Inside the if-condition, only `alpha` is given to test if `alpha` is nonzero or not NULL.

```
#include <stdio.h>
```

```
void assign_and_print(int alpha, int beta)
{
    alpha = beta;
    if(alpha)
    {
        printf("%d", alpha);
    }
}
```

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

EXP45-C

# CERT C: Rule EXP46-C

Do not use a bitwise operator with a Boolean-like operand

## Description

### Rule Definition

*Do not use a bitwise operator with a Boolean-like operand.*

### Polyspace Implementation

The rule checker checks for **Use of bitwise operator with a Boolean-like operand**.

## Examples

### Use of bitwise operator with a Boolean-like operand

#### Issue

**Use of bitwise operator with a Boolean-like operand** occurs when you use bitwise operators, such as:

- Bitwise AND (&, &=)
- Bitwise OR (|, |=)
- Bitwise XOR (^, ^=)
- Bitwise NOT(~)

with:

- Boolean type variables
- Outputs of relational or equality expressions

Using Boolean type variables as array indices, in Boolean arithmetic expression, and in shifting operations does not raise this defect.

#### Risk

Boolean-like operands, such as variables of type `bool` and outputs of relational operators typically appear in logical expressions. Using a bitwise operator in an expression containing Boolean variables and relational operators might be a sign of logic error. Because bitwise operators and logical operators look similar, you might inadvertently use a bitwise operator instead of a logical operator. Such logic errors do not raise any compilation error and can introduce bugs in your code that are difficult to find.

#### Fix

Use logical operators in expressions that contain Boolean variables and relational operator. To indicate that you intend to use a bitwise operator in such an expression, use parentheses.

#### Example — Possible Bug Due to Using Bitwise Operator

```
int getuid();
int geteuid();
```

```
void Noncompliant ()
{
    if (getuid () & geteuid () == 0) {//Noncompliant
        /* ... */
    }else{
        /*...*/
    }
}
```

In this example, the `if-else` block is executed conditionally. The conditional statement uses the bitwise AND (&) instead of the logical AND (&&), perhaps by mistake. Consider when the function `geteuid()` evaluates to 0, and `getuid()` evaluates to 2. In this case, the `else` block of code executes if you use & because `2&1` evaluates to `false`. Conversely, the `if` block of code executes when you use && because `2&&1` evaluates to `true`. Using & instead of && might introduce logic errors and bugs in your code that are difficult to find. Polyspace flags the use of bitwise operators in these kinds of expressions where relational operators are also used.

#### Correction — Use Logical Operators with Boolean-Like Operands

One possible correction is to use logical operators in expressions that contain relational operators and Boolean variables.

```
int getuid();
int geteuid();
void Compliant ()
{
    if (getuid () && geteuid () == 0) {
        /* ... */
    }else{
        /*...*/
    }
}
```

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

EXP46-C

# CERT C: Rule EXP47-C

Do not call va_arg with an argument of the incorrect type

## Description

### Rule Definition

*Do not call va_arg with an argument of the incorrect type.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Incorrect data type passed to va_arg**.
- **Too many va_arg calls for current argument list**.

## Examples

### Incorrect data type passed to va_arg

#### Issue

**Incorrect data type passed to va_arg** when the data type in a va_arg call does not match the data type of the variadic function argument that va_arg reads.

For instance, you pass an unsigned char argument to a variadic function func. Because of default argument promotion, the argument is promoted to int. When you use a va_arg call that reads an unsigned char argument, a type mismatch occurs.

```
void func (int n, ...) {
   //...
   va_list args;
   va_arg(args, unsigned char);
   //...
}

void main(void) {
   unsigned char c;
   func(1,c);
}
```

#### Risk

In a variadic function (function with variable number of arguments), you use va_arg to read each argument from the variable argument list (va_list). The va_arg use does not guarantee that there actually exists an argument to read or that the argument data type matches the data type in the va_arg call. You have to make sure that both conditions are true.

Reading an incorrect type with a va_arg call can result in undefined behavior. Because function arguments reside on the stack, you might access an unwanted area of the stack.

#### Fix

Make sure that the data type of the argument passed to the variadic function matches the data type in the `va_arg` call.

Arguments of a variadic function undergo default argument promotions. The argument data types of a variadic function cannot be determined from a prototype. The arguments of such functions undergo default argument promotions (see Sec. 6.5.2.2 and 7.15.1.1 in the C99 Standard). Integer arguments undergo integer promotion and arguments of type `float` are promoted to `double`. For integer arguments, if a data type can be represented by an `int`, for instance, `char` or `short`, it is promoted to an `int`. Otherwise, it is promoted to an `unsigned int`. All other arguments do not undergo promotion.

To avoid undefined and implementation-defined behavior, minimize the use of variadic functions. Use the checkers for `MISRA C:2012 Rule 17.1` or `MISRA C++:2008 Rule 8-4-1` to detect use of variadic functions.

#### Example - `char` Used as Function Argument Type and va_arg argument

```
#include <stdarg.h>
#include <stdio.h>

unsigned char func(size_t count, ...) {
    va_list ap;
    unsigned char result = 0;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, unsigned char); //Noncompliant
    }
    va_end(ap);
    return result;
}

void func_caller(void) {
    unsigned char c = 0x12;
    (void)func(1, c);
}
```

In this example, `func` takes an `unsigned char` argument, which undergoes default argument promotion to `int`. The data type in the `va_arg` call is still `unsigned char`, which does not match the `int` argument type.

#### Correction — Use `int` as va_arg Argument

One possible correction is to read an `int` argument with `va_arg`.

```
#include <stdarg.h>
#include <stdio.h>

unsigned char func(size_t count, ...) {
    va_list ap;
    unsigned char result = 0;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
    }
    va_end(ap);
    return result;
}

void func_caller(void) {
    unsigned char c = 0x12;
    (void)func(1, c);
}
```

### Too many va_arg calls for current argument list

#### Issue

**Too many va_arg calls for current argument list** occurs when the number of calls to va_arg exceeds the number of arguments passed to the corresponding variadic function. The analysis raises a defect only when the variadic function is called.

**Too many va_arg calls for current argument list** does not raise a defect when:

- The number of calls to va_arg inside the variadic function is indeterminate. For example, if the calls are from an external source.
- The va_list used in va_arg is invalid.

#### Risk

When you call va_arg and there is no next argument available in va_list, the behavior is undefined. The call to va_arg might corrupt data or return an unexpected result.

#### Fix

Ensure that you pass the correct number of arguments to the variadic function.

#### Example - No Argument Available When Calling va_arg

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/
int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {
/* No further argument available
* in va_list when calling va_arg
*/

            result += va_arg(ap, int); //Noncompliant
        }
    }
    va_end(ap);
    return result;
}
```

```
void func(void) {

    (void)variadic_func(2, 100);

}
```

In this example, the named argument and only one variadic argument are passed to `variadic_func()` when it is called inside `func()`. On the second call to `va_arg`, no further variadic argument is available in `ap` and the behavior is undefined.

#### Correction — Pass Correct Number of Arguments to Variadic Function

One possible correction is to ensure that you pass the correct number of arguments to the variadic function.

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/

int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {

/* The correct number of arguments is
* passed to va_list when variadic_func()
* is called inside func()
*/
            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100, 200);

}
```

## Check Information

**Group:** Rule 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP47-C

# CERT C: Rule INT30-C

Ensure that unsigned integer operations do not wrap

## Description

### Rule Definition

*Ensure that unsigned integer operations do not wrap.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Unsigned integer overflow**.
- **Unsigned integer constant overflow**.

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Unsigned integer overflow

#### Issue

**Unsigned integer overflow** occurs when an operation on unsigned integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

The C11 standard states that unsigned integer overflows result in wrap-around behavior. However, a wrap around behavior might not always be desirable. For instance, if the result of a computation is used as an array size and the computation overflows, the array size is much smaller than expected.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.

- Checking for values that lead to the overflow and performing appropriate error handling. In the error handling code, you can override the default wrap-around behavior for overflows and implement saturation behavior, for instance.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Add One to Maximum Unsigned Integer

```
#include <limits.h>

unsigned int plusplus(void) {

    unsigned uvar = UINT_MAX;
    uvar++; //Noncompliant
    return uvar;
}
```

In the third statement of this function, the variable `uvar` is increased by 1. However, the value of `uvar` is the maximum unsigned integer value, so 1 plus the maximum integer value cannot be represented by an `unsigned int`. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `uvar` is reduced by modulo `UINT_MAX`. The result is `uvar = 1`.

#### Correction — Different Storage Type

One possible correction is to store the operation result in a larger data type. In this example, by returning an `unsigned long long` instead of an `unsigned int`, the overflow error is fixed.

```
#include <limits.h>

unsigned long long plusplus(void) {

    unsigned long long ullvar = UINT_MAX;
    ullvar++;
    return ullvar;
}
```

### Unsigned integer constant overflow

#### Issue

**Unsigned integer constant overflow** occurs when you assign a compile-time constant to a unsigned integer variable whose data type cannot accommodate the value. An `n`-bit unsigned integer holds values in the range `[0, 2`$^n$`-1]`.

For instance, `c` is an 8-bit unsigned `char` variable that cannot hold the value 256.

```
unsigned char c = 256;
```

To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.

**Risk**

The C standard states that overflowing unsigned integers must be wrapped around (see, for instance, the C11 standard, section 6.2.5). However, the wrap-around behavior can be unintended and cause unexpected results.

**Fix**

Check if the constant value is what you intended. If the value is correct, use a wider data type for the variable.

**Example - Overflowing Constant from Macro Expansion**

```
#define MAX_UNSIGNED_CHAR 255 //Noncompliant
#define MAX_UNSIGNED_SHORT 65535 //Noncompliant

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned short c2 = MAX_UNSIGNED_SHORT + 1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow.

**Correction — Use Wider Data Type**

One possible correction is to use a wider data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned short c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned int c2 = MAX_UNSIGNED_SHORT + 1;
}
```

## Check Information

**Group:** Rule 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

INT30-C

# CERT C: Rule INT31-C

Ensure that integer conversions do not result in lost or misinterpreted data

## Description

### Rule Definition

*Ensure that integer conversions do not result in lost or misinterpreted data.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Integer conversion overflow**.
- **Call to memset with unintended value**.
- **Sign change integer conversion overflow**.
- **Tainted sign change conversion**.
- **Unsigned integer conversion overflow**.

### Extend Checker

Extend this checker to run a stricter analysis:

- When the input values are unknown and only a subset of inputs causes an issue, a default Bug Finder analysis might not detect an **Integer conversion overflow**, **Sign change integer conversion overflow**, or **Unsigned integer conversion overflow**. To check for these issues caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.
- A default Bug Finder analysis might not flag a **Tainted sign change conversion** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Integer conversion overflow

#### Issue

**Integer conversion overflow** occurs when converting an integer to a smaller integer type. If the variable does not have enough bytes to represent the original value, the conversion overflows. For instance, if you perform a comparison between implementation-defined type `time_t` and a signed integer, Polyspace reports a violation because `time_t` might be implemented as an unsigned integer.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer conversion overflows result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Converting from `int` to `char`

```
char convert(void) {

    int num = 1000000;

    return (char)num; //Noncompliant
}
```

In the return statement, the integer variable `num` is converted to a `char`. However, an 8-bit or 16-bit character cannot represent 1000000 because it requires at least 20 bits. So the conversion operation overflows.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number.

```
long convert(void) {

    int num = 1000000;

    return (long)num;
}
```

### Call to memset with unintended value

#### Issue

**Call to memset with unintended value** occurs when Polyspace Bug Finder detects a use of the `memset` or `wmemset` function with possibly incorrect arguments.

`void *memset (void *ptr, int value, size_t num)` fills the first `num` bytes of the memory block that `ptr` points to with the specified `value`. If the argument `value` is incorrect, the memory block is initialized with an unintended value.

The unintended initialization can occur in the following cases.

| Issue | Risk | Possible Fix |
|---|---|---|
| The second argument is `'0'` instead of `0` or `'\0'`. | The ASCII value of character `'0'` is `48` (decimal), `0x30` (hexadecimal), `069` (octal) but not `0` (or `'\0'`) . | If you want to initialize with `'0'`, use one of the ASCII values. Otherwise, use `0` or `'\0'`. |
| The second and third arguments are probably reversed. For instance, the third argument is a literal and the second argument is not a literal. | If the order is reversed, a memory block of unintended size is initialized with incorrect arguments. | Reverse the order of the arguments. |
| The second argument cannot be represented in a byte. | If the second argument cannot be represented in a byte, and you expect each byte of a memory block to be filled with that argument, the initialization does not occur as intended. | Apply a bit mask to the argument to produce a wrapped or truncated result that can be represented in a byte. When you apply a bit mask, make sure that it produces an expected result.<br>For instance, replace `memset(a, -13, sizeof(a))` with `memset(a, (-13) & 0xFF, sizeof(a))`. |

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Value Cannot Be Represented in a Byte**

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE];
    int c = -2;
    memset(buf, (char)c, sizeof(buf)); //Noncompliant
}
```

In this example, `(char)c` cannot be represented in a byte.

**Correction — Apply Cast**

One possible correction is to apply a cast so that the result can be represented in a byte. However, check that the result of the cast is an acceptable initialization value. Polyspace still flags the call to `memset` because the casting the signed integer c to an unsigned character overflows.

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE   ];
    int c = -2;
    memset(buf, (unsigned char)c, sizeof(buf));  //Noncompliant
}
```

**Correction — Avoid Using `memset`**

One possible correction is to reserve the use of `memset` only for setting or clearing all bits in a buffer. For instance, in this code, `memset` is called to clear the bits of the character array `buf`.

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE   ];
    int c = -2;
    memset(buf, 0, sizeof(buf));//Compliant
    /* After clearing buf, use it in operations*/
}
```

### Sign change integer conversion overflow

**Issue**

**Sign change integer conversion overflow** occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

**Fix**

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do

not show the event history, you can trace back using right-click options in the source code and see previous related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface".

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Convert from `unsigned char` to `char`

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count; //Noncompliant
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

#### Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

### Tainted sign change conversion

#### Issue

**Tainted sign change conversion** looks for values from unsecure sources that are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
memmove
strncmp
strncpy
calloc
malloc
memalign
```

#### Risk

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

#### Fix

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

#### Example - Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(int size) {
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size);  //Noncompliant
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

#### Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(int size) {
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

### Unsigned integer conversion overflow

#### Issue

**Unsigned integer conversion overflow** occurs when converting an unsigned integer to a smaller unsigned integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer conversion overflows result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Converting from `int` to `char`

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum;   //Noncompliant
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned long)unum;
}
```

## Check Information

**Group:** Rule 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

INT31-C

# CERT C: Rule INT32-C

Ensure that operations on signed integers do not result in overflow

## Description

### Rule Definition

*Ensure that operations on signed integers do not result in overflow.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Integer overflow**.
- **Tainted division operand**.
- **Tainted modulo operand**.

### Extend Checker

When the input values are unknown and only a subset of inputs causes an issue, a default Bug Finder analysis might not detect an **Integer overflow**. To check for these issues caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Integer overflow

#### Issue

**Integer overflow** occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`. W

#### Risk

Integer overflows on signed integers result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;              //Noncompliant
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

#### Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

### Tainted division operand

#### Issue

**Tainted division operand** detects division operations where one or both of the integer operands is from an unsecure source.

#### Risk

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

#### Fix

Before performing the division, validate the values of the operands. Check for denominators of `0` or `-1`, and numerators of the minimum integer value.

#### Example - Division of Function Arguments

```
#include <limits.h>
#include <stdio.h>

extern void print_int(int);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    int r =  num/denum;  //Noncompliant
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

#### Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include <limits.h>
#include <stdio.h>

extern void print_long(long);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    long res= 0;
    if (denum!=0 && !(num==INT_MIN && denum==-1)) {
        res = num/denum;
    }
    print_long(res);
    return res;
}
```

### Tainted modulo operand

#### Issue

**Tainted modulo operand** checks the operands of remainder `%` operations. Bug Finder flags modulo operations with one or more tainted operands.

#### Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

#### Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of `0` and `-1`. Check both operands for negative values.

#### Example - Modulo with Unsecure Input

```
#include<stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d",&userden);
    int rem =  128%userden;  //Noncompliant
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using a user input. The user input is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

#### Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
#include<stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d",&userden);
    int rem = 0;
    if (userden > 0) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

## Check Information

**Group:** Rule 04. Integers (INT)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

INT32-C

# CERT C: Rule INT33-C

Ensure that division and remainder operations do not result in divide-by-zero errors

## Description

### Rule Definition

*Ensure that division and remainder operations do not result in divide-by-zero errors.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Integer division by zero**.
- **Tainted division operand**.
- **Tainted modulo operand**.

### Extend Checker

Extend this checker to run a stricter analysis:

- When the input values are unknown and only a subset of inputs causes an issue, a default Bug Finder analysis might not detect an **Integer division by zero**. To check for these issues caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.
- A default Bug Finder analysis might not flag a **Tainted division operand** or **Tainted modulo operand** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Integer division by zero

#### Issue

**Integer division by zero** occurs when the denominator of a division or modulo operation can be a zero-valued integer.

#### Risk

A division by zero can result in a program crash.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the denominator variable acquires a zero value. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    result = num/denom; //Noncompliant

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

#### Correction — Check Before Division

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

#### Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2;
    int result = 0;
    result = num/denom;

    return result;
}
```

#### Example - Modulo Operation with Zero

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        arr[i] = input % i; //Noncompliant
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

In this example, Polyspace flags the modulo operation as a division by zero. Because modulo is inherently a division operation, the divisor (right hand argument) cannot be zero. The modulo operation uses the `for` loop index as the divisor. However, the `for` loop starts at zero, which cannot be an iterator.

#### Correction — Check Divisor Before Operation

One possible correction is checking the divisor before the modulo operation. In this example, see if the index `i` is zero before the modulo operation.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        if(i != 0)
        {
             arr[i] = input % i;
        }
        else
        {
             arr[i] = input;
        }
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

#### Correction — Change Divisor

Another possible correction is changing the divisor to a nonzero integer. In this example, add one to the index before the `%` operation to avoid dividing by zero.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
         arr[i] = input % (i+1);
    }
    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

### Tainted division operand

#### Issue

**Tainted division operand** detects division operations where one or both of the integer operands is from an unsecure source.

#### Risk

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

#### Fix

Before performing the division, validate the values of the operands. Check for denominators of `0` or `-1`, and numerators of the minimum integer value.

#### Example - Division of Function Arguments

```
#include<stdio.h>
extern void print_int(int);

int taintedintdivision(void) {
    int num, den;
    scanf("%lf %lf",&num, &den);
    int r =  num/den;  //Noncompliant
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

#### Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include<stdio.h>
#include <limits.h>
extern void print_int(int);

int taintedintdivision(void) {
    int num, den;
    scanf("%lf %lf",&num, &den);
    int r = 0;
    if (den!=0 && !(num=INT_MIN && den==-1)){
        r =  num/den;
    }
    print_int(r);
    return r;
}
```

### Tainted modulo operand

#### Issue

**Tainted modulo operand** checks the operands of remainder `%` operations. Bug Finder flags modulo operations with one or more tainted operands.

#### Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

#### Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of `0` and `-1`. Check both operands for negative values.

#### Example - Modulo with UInsecure Input

```
#include<stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d",&userden);
    int rem =  128%userden;  //Noncompliant
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using a user input. The input is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

#### Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
#include<stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d",&userden);
    int rem = 0;
    if (userden > 0) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

## Check Information

**Group:** Rule 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

INT33-C

# CERT C: Rule INT34-C

Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand

## Description

### Rule Definition

*Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Shift of a negative value**.
- **Shift operation overflow**.

### Extend Checker

When the input values are unknown and only a subset of inputs causes an issue, a default Bug Finder analysis might not detect a **Shift of a negative value** or **Shift operation overflow**. To check for these issues caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Shift of a negative value

#### Issue

**Shift of a negative value** occurs when a bit-wise shift is used on a variable that can have negative values.

#### Risk

Shifts on negative values overwrite the sign bit that identifies a number as negative. The shift operation can result in unexpected values.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being shifted acquires negative values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

To fix the defect, check for negative values before the bit-wise shift operation and perform appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Shifting a negative variable

```
int shifting(int val)
{
    int res = -1;
    return res << val; //Noncompliant
}
```

In the return statement, the variable `res` is shifted a certain number of bits to the left. However, because `res` is negative, the shift might overwrite the sign bit.

#### Correction — Change the Data Type

One possible correction is to change the data type of the shifted variable to unsigned. This correction eliminates the sign bit, so left shifting does not change the sign of the variable.

```
int shifting(int val)
{
    unsigned int res = -1;
    return res << val;
}
```

### Shift operation overflow

#### Issue

**Shift operation overflow** occurs when a shift operation can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Shift operation overflows can result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the shift operation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface".

You can fix the defect by:

- Using a bigger data type for the result of the shift operation so that all values can be accommodated.

- Checking for values that lead to the overflow and performing appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example - Left Shift of Integer**

```
int left_shift(void) {

    int foo = 33;
    return 1 << foo;  //Noncompliant
}
```

In the return statement of this function, bit-wise shift operation is performed shifting 1 `foo` bits to the left. However, an `int` has only 32 bits, so the range of the shift must be between 0 and 31. Therefore, this shift operation causes an overflow.

**Correction — Different storage type**

One possible correction is to store the shift operation result in a larger data type. In this example, by returning a `long long` instead of an `int`, the overflow defect is fixed.

```
long long left_shift(void) {

    int foo = 33;
    return 1LL << foo;
}
```

## Check Information

**Group:** Rule 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

INT34-C

# CERT C: Rule INT35-C

Use correct integer precisions

## Description

### Rule Definition

*Use correct integer precisions.*

### Polyspace Implementation

The rule checker checks for **Integer precision exceeded**.

## Examples

### Integer precision exceeded

#### Issue

**Integer precision exceeded** occurs when an integer expression uses the integer size in an operation that exceeds the integer precision. On some architectures, the size of an integer in memory can include sign and padding bits. On these architectures, the integer size is larger than the precision which is just the number of bits that represent the value of the integer.

#### Risk

Using the size of an integer in an operation on the integer precision can result in integer overflow, wrap around, or unexpected results. For instance, an unsigned integer can be stored in memory in 64 bits, but uses only 48 bits to represent its value. A 56 bits left-shift operation on this integer is undefined behavior.

Assuming that the size of an integer is equal to its precision can also result in program portability issues between different architectures.

#### Fix

Do not use the size of an integer instead of its precision. To determine the integer precision, implement a precision computation routine or use a builtin function such as `__builtin_popcount()`.

#### Example - Using Size of `unsigned int` for Left Shift Operation

```
#include <limits.h>

unsigned int func(unsigned int exp)
{
    if (exp >= sizeof(unsigned int) * CHAR_BIT) {
        /* Handle error */
    }
    return 1U << exp; //Noncompliant
}
```

In this example, the function uses a left shift operation to return the value of 2 raised to the power of `exp`. The operation shifts the bits of `1U` by `exp` positions to the left. The `if` statement ensures that the operation does not shift the bits by a number of positions `exp` greater than the size of an `unsigned int`. However, if `unsigned int` contains padding bits, the value returned by `sizeof()` is larger than the precision of `unsigned int`. As a result, some values of `exp` might be too large, and the shift operation might be undefined behavior.

#### Correction — Implement Function to Compute Precision of `unsigned int`

One possible correction is to implement a function `popcount()` that computes the precision of `unsigned int` by counting the number of set bits.

```
#include <stddef.h>
#include <stdint.h>
#include <limits.h>

size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)


unsigned int func(unsigned int exp)
{
    if (exp >= PRECISION(UINT_MAX)) {
        /* Handle error */
    }
    return 1 << exp;
}




size_t popcount(uintmax_t num)
{
    size_t precision = 0;
    while (num != 0) {
        if (num % 2 == 1) {
            precision++;
        }
        num >>= 1;
    }
    return precision;
}
```

## Check Information

**Group:** Rule 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

INT35-C

# CERT C: Rule INT36-C

Converting a pointer to integer or integer to pointer

## Description

### Rule Definition

*Converting a pointer to integer or integer to pointer.*

### Polyspace Implementation

The rule checker checks for **Conversion between pointer and integer**

## Examples

### Conversion between pointer and integer

#### Issue

**Conversion between pointer and integer** is raised when any of these conditions is true:

- A pointer type is converted into an integral type with smaller size. When converting a pointer of the types `intptr_t` or `uintprt_t` into integral types `enum`, `ptrdiff_t`, or `pid_t`, Polyspace raises a defect if the size of pointers and integer types are different in your environment. In `i386` environment, both pointers and integer types have a size of 32 bits. In this environment, Polyspace does not flag conversion from a pointer to an integer of same size. But in `x86_64` environment where pointers are 64 bits and unsigned integers are 32 bits, Polyspace flags conversion from pointers to integers of different sizes.
- An integral type is converted into a raw pointer type. Polyspace flags conversion from integral types to raw pointers regardless of their sizes. For instance, if an integer originates from a pointer and then later, is cast into a pointer, Polyspace flags the conversion from integer to pointer. In this case, the integer and the pointer are of the same size, but the conversion is still flagged.

#### Risk

The mapping between pointers and integers is not always consistent with the addressing structure of the environment.

Converting from pointers to integers can create:

- Truncated or out of range integer values.
- Invalid integer types.

Converting from integers to pointers can create:

- Misaligned pointers or misaligned objects.
- Invalid pointer addresses.

### Fix

Where possible, avoid pointer-to-integer or integer-to-pointer conversions. If you want to convert a `void` pointer to an integer, so that you do not change the value, use types:

- C99 — `intptr_t` or `uintptr_t`
- C90 — `size_t` or `ssize_t`

### Example — Integer to Pointer Conversions

```
unsigned int* badintptrcast(void)
{
    int* ptr;
    unsigned long long int_same_as_ptr ;
    unsigned int int_smaller_than_ptr;

    unsigned int* ptr0 = (unsigned int*)0xdeadbeef; //Noncompliant


    /*int to ptr of same size*/
    ptr = (int*)int_same_as_ptr;  //Noncompliant

    /*int to ptr of different size*/
    ptr = (int*)int_smaller_than_ptr;  //Noncompliant

    return (unsigned int*)(ptr0 - (unsigned int*)ptr);  //Noncompliant
}
```

In this example, Polyspace flags the conversions that might be unsafe. For instance:

- The conversion of `0xdeadbeef` to `unsigned int*` causes alignment issues for the pointer. Polyspace flags the conversion.
- The conversion of the `int_same_as_ptr` and `int_smaller_than_ptr` to `int*` pointers might result in invalid pointer address. Polyspace flags these conversion.
- The `return` statement casts `ptrdiff_t` to a pointer. This pointer might not point to an invalid address. Polyspace flags the conversion.

### Correction — Use `intptr_t`

One possible correction is to use `intptr_t` types to store addresses. When storing the result of subtracting two pointers, use the type `ptrdiff_t`.

```
#include<stdint.h>
#include<stddef.h>
ptrdiff_t badintptrcast(void)
{
    intptr_t ptr;
    unsigned long long int_same_as_ptr ;
    unsigned int int_smaller_than_ptr;

    intptr_t ptr0 = (intptr_t)0xdeadbeef; //Compliant

    /*int to ptr of same size*/
    ptr = (intptr_t)int_same_as_ptr;  //Compliant

    /*int to ptr of different size*/
    ptr = (intptr_t)int_smaller_than_ptr;  //Compliant
    int offset = 0;
    return (ptrdiff_t)(ptr0 - offset);//Compliant

}
```

## Check Information

**Group:** Rule 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

INT36-C

# CERT C: Rule FLP30-C

Do not use floating-point variables as loop counters

## Description

### Rule Definition

*Do not use floating-point variables as loop counters.*

### Polyspace Implementation

The rule checker checks for **Use of float variable as loop counter**.

## Examples

### Use of float variable as loop counter

#### Issue

The issue occurs when a loop counter has a floating type.

If the `for` index is a variable symbol, Polyspace checks that it is not a float.

#### Risk

When using a floating-point loop counter, accumulation of rounding errors can result in a mismatch between the expected and actual number of iterations. This rounding error can happen when a loop step that is not a power of the floating point radix is rounded to a value that can be represented by a float.

Even if a loop with a floating-point loop counter appears to behave correctly on one implementation, it can give a different number of iteration on another implementation.

#### Example - `for` Loop Counters

In this example, the three `for` loops show three different loop counters. The first and second `for` loops use float variables as loop counters, and therefore are not compliant. The third loop uses the integer `count` as the loop counter. Even though `count` is used as a float inside the loop, the variable remains an integer when acting as the loop index. Therefore, this `for` loop is compliant.

```
int main(void){
    unsigned int counter = 0u;
    int result = 0;
    float foo;

    // Float loop counters
    for(float foo = 0.0f; foo < 1.0f; foo +=0.001f){ /*Non-compliant*/
        /*counter = 1000 at the end of the loop */
        ++counter;
    }

    float fff = 0.0f;
    for(fff = 0.0f; fff <12.0f; fff += 1.0f){    /* Non-compliant*/
        result++;
    }

    // Integer loop count
    for(unsigned int count = 0u; count < 1000u; ++count){ /* Compliant */
        foo = (float) count * 0.001f;
    }
}
```

### `while` Loop Counters

This example shows two `while` loops both of which use floating point variables in the `while`-loop conditions:

- The first `while` loop uses the floating point variable `foo` in the condition and inside the loop. Because `foo` changes, floating-point rounding errors can cause unexpected behavior. Polyspace reports a violation.
- In the second while loop, the floating point array `buffer` is used in the loop condition. Polyspace identifies `iter1` and `iter2` as the loop variable. Because the loop variables are not floating point variables, a violation is not reported.

```
int main(void){
    unsigned int iter1 =0;
    int iter2;
    float foo;
    double buffer[2];
    double tmp;

    foo = 0.0f;
    while (foo < 1.0f){/* Non-compliant - foo used as a loop counter */
        foo += 0.001f;
    }

    //...
    while((iter1+1 < 2)&& (buffer[iter1]<buffer[iter2])){ //Compliant - loop counter is integer
        // swap buffer[iter1] and buffer[iter2]
        tmp = buffer[iter2];
        buffer[iter2] = buffer[iter1];
        buffer[iter1] = tmp;
        iter2 = iter1;
        iter1++;
    }

    return 1;
}
```

## Check Information

**Group:** Rule 05. Floating Point (FLP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FLP30-C

# CERT C: Rule FLP32-C

Prevent or detect domain and range errors in math functions

## Description

### Rule Definition

*Prevent or detect domain and range errors in math functions.*

### Polyspace Implementation

The rule checker checks for **Invalid use of standard library floating point routine**.

### Extend Checker

Extend this checker to check for defects caused by specific values and invalid use of functions from a custom library. For instance:

- You might be using a custom library of mathematical floating point functions. If a custom library function have the same domain and range as another function from the standard library, you can extend this checker to check the custom library function. See “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries”.
- A default Bug Finder analysis might not raise a defect when the input values are unknown and only a subset of inputs can cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Invalid use of standard library floating point routine

#### Issue

**Invalid use of standard library floating point routine** occurs when you use invalid arguments with a floating point function from the standard library. This defect picks up:

- Rounding and absolute value routines

  `ceil, fabs, floor, fmod`
- Fractions and division routines

  `fmod, modf`
- Exponents and log routines

  `frexp, ldexp, sqrt, pow, exp, log, log10`
- Trigonometry function routines

  `cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh, atanh`

#### Risk

Domain errors on standard library floating point functions result in implementation-defined values. If you use the function return value in subsequent computations, you can see unexpected results.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the function argument acquires invalid values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface".

It is a good practice to handle for domain errors before using a standard library floating point function. For instance, before calling the `acos` function, check if the argument is in [-1.0, 1.0] and handle the error.

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and `NaNs`. Operations that results in infinities and `NaNs` might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats)`.

#### Example – Domain Error

```
#include <math.h>

double arccosine(void) {
    double degree = 5.0;
    return acos(degree); //Noncompliant
}
```

The input value to `acos` must be in the interval `[-1,1]`. This input argument, `degree`, is outside this range.

#### Correction – Change Input Argument

One possible correction is to change the input value to fit the specified range. In this example, change the input value from degrees to radians to fix this defect.

```
#include <math.h>

double arccosine(void) {
    double degree = 5.0;
    double radian = degree * 3.14159 / 180.;
    return acos(radian);
}
```

**Example – Range Error**

```
#include <math.h>
#include <fenv.h>
#include <errno.h>

void calculate_sinh (double x)
{
    double result;
    result = sinh (x);   //Noncompliant
}

void call_sinh_calculator()
{
    double x = 10.e32;
    calculate_sinh(x);
}
```

The `sinh` function is called with a large argument that results in an overflow.

**Correction – Handle Range Errors**

You cannot prevent range errors but only handle them later.

For instance, if a range error occurs, the function `sinh` sets `errno` to nonzero values on certain implementations. In those implementations, you can check `errno` for non-zero values to handle error conditions.

The following function uses the error handling recommendations given in FLP32-C.

```
#include <math.h>
#include <fenv.h>
#include <errno.h>

void calculate_sinh (double x)
{
    double result;
    result = sinh (x);   //Noncompliant
    if ((math_errhandling & MATH_ERRNO) && errno != 0) {
        /* Handle range error */
    }
    else if ((math_errhandling & MATH_ERREXCEPT) && fetestexcept (FE_INVALID | FE_DIVBYZERO | FE_
        /* Handle range error */
    }
}

void call_sinh_calculator()
{
    double x = 10.e32;
    calculate_sinh(x);
}
```

Note that the rule checker continues to report the range error even if it is handled later. In this case, you can mark the error as justified. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

## Check Information

**Group:** Rule 05. Floating Point (FLP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FLP32-C

# CERT C: Rule FLP34-C

Ensure that floating-point conversions are within range of the new type

## Description

### Rule Definition

*Ensure that floating-point conversions are within range of the new type.*

### Polyspace Implementation

This checker checks for:

- **Float conversion overflow**
- **Floating point to integer conversion overflow**

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Float conversion overflow

#### Issue

**Float conversion overflow** occurs when converting a floating point value to a smaller floating point data type. If the variable does not have enough memory to represent the original value, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing conversion in subsequent computations and do not account for the overflow, you can see unexpected results.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being converted acquires its current value You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller floating point types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and `NaN`s. Operations that results in infinities and `NaN`s might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats)`.

#### Example - Converting from `double` to `float`

```
float convert(void) {

    double diam = 1e100;
    return (float)diam; //Noncompliant
}
```

In the return statement, the variable `diam` of type double (64 bits) is converted to a variable of type float (32 bits). However, the value 1^100 requires more than 32 bits to be precisely represented.

### Floating point to integer conversion overflow

#### Issue

**Floating point to integer conversion overflow** occurs when converting a floating-point value to an integer data type. If the integer part of the value cannot be represented within the storage available for the integer data type, the conversion overflows.

#### Risk

When converting from floating point to integer types, if the floating point value is outside the range that can be represented by the integer type, the behavior is undefined (C Standard 6.3.14 and 6.3.15).

#### Fix

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

  A check for overflowing values on a `float` variable `var` can be like this:

```
if  isnan(var)
    || popcount(INT_MAX) < log2f(fabsf(var))
    || (var != 0.0F && fabsf(var) < FLT_MIN)){
        // Handle error
    }
    else {
        // Perform operations on var
    }
```

The check determines if the floating point value is representable within an integer type:

- The value is not NaN.
- The number of bits required to store the value is less than the number of bits in `INT_MAX` (the largest integer that the `int` type can represent). The `popcount` function (not defined here) counts the number of 1's (or set bits) in a number.
- The floating point value is not lower than the smallest representable floating-point value.

**Example – Floating Point Value Converted to Integer Without Handling Overflows**

```
void func(float fVar) {
  int iVar;
  iVar = fVar; //Noncompliant //Noncompliant
}
```

In this example, the floating point value of `fVar` is not checked for overflows before converting to an integer type. Since the argument `fVar` can contain values that are not representable within the `int` data type, the analysis flags a potential overflow.

Note that `func` is not called in this example, and the overflow is only a possibility. To see issues of these types, add the analysis option `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`.

## Check Information

**Group:** Rule 05. Floating Point (FLP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FLP34-C

# CERT C: Rule FLP36-C

Preserve precision when converting integral values to floating-point type

## Description

### Rule Definition

*Preserve precision when converting integral values to floating-point type.*

### Polyspace Implementation

The rule checker checks for **Precision loss in integer to float conversion**.

## Examples

### Precision loss in integer to float conversion

#### Issue

**Precision loss from integer to float conversion** occurs when you cast an integer value to a floating-point type that cannot represent the original integer value.

For instance, the `long int` value `1234567890L` is too large for a variable of type `float` .

#### Risk

If the floating-point type cannot represent the integer value, the behavior is undefined (see C11 standard, 6.3.1.4, paragraph 2). For instance, least significant bits of the variable value can be dropped leading to unexpected results.

#### Fix

Convert to a floating-point type that can represent the integer value.

For instance, if the `float` data type cannot represent the integer value, use the `double` data type instead.

When writing a function that converts an integer to floating point type, before the conversion, check if the integer value can be represented in the floating-point type. For instance, `DBL_MANT_DIG * log2(FLT_RADIX)` represents the number of base-2 digits in the type `double`. Before conversion to the type `double`, check if this number is greater than or equal to the precision of the integer that you are converting. To determine the precision of an integer `num`, use this code:

```
size_t precision = 0;
while (num != 0) {
   if (num % 2 == 1) {
     precision++;
   }
   num >>= 1;
}
```

Some implementations provide a builtin function to determine the precision of an integer. For instance, GCC provides the function `__builtin_popcount`.

**Example - Conversion of Large Integer to Floating-Point Type**

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  float approx = big; //Noncompliant
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

In this example, the `long int` variable `big` is converted to `float`.

**Correction — Use a Wider Floating-Point Type**

One possible correction is to convert to the `double` data type instead of `float`.

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  double approx = big;
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

## Check Information

**Group:** Rule 05. Floating Point (FLP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FLP36-C

# CERT C: Rule FLP37-C

Do not use object representations to compare floating-point values

## Description

### Rule Definition

*Do not use object representations to compare floating-point values.*

### Polyspace Implementation

The rule checker checks for **Memory comparison of float-point values**.

## Examples

### Memory comparison of float-point values

**Issue**

**Memory comparison of float-point values** occurs when you compare the object representation of floating-point values or the object representation of structures containing floating-point members. When you use the functions `memcmp`, `bcmp`, or `wmemcmp` to perform the bit pattern comparison, the defect is raised.

**Risk**

The object representation of floating-point values uses specific bit patterns to encode those values. Floating-point values that are equal, for instance `-0.0` and `0.0` in the IEC 60559 standard, can have different bit patterns in their object representation. Similarly, floating-point values that are not equal can have the same bit pattern in their object representation.

**Fix**

When you compare structures containing floating-point members, compare the structure members individually.

To compare two floating-point values, use the `==` or `!=` operators. If you follow a standard that discourages the use of these operators, such as MISRA, ensure that the difference between the floating-point values is within an acceptable range.

**Example - Using `memcmp` to Compare Structures with Floating-Point Members**

```
#include <string.h>

typedef struct {
    int i;
    float f;
} myStruct;

extern void initialize_Struct(myStruct *);

int func_cmp(myStruct *s1, myStruct *s2) {
/* Comparison between structures containing
* floating-point members */
    return memcmp  //Noncompliant
        ((const void *)s1, (const void *)s2, sizeof(myStruct));
}

void func(void) {
    myStruct s1, s2;
    initialize_Struct(&s1);
    initialize_Struct(&s2);
    (void)func_cmp(&s1, &s2);
}
```

In this example, `func_cmp()` calls `memcmp()` to compare the object representations of structures `s1` and `s2`. The comparison might be inaccurate because the structures contain floating-point members.

#### Correction — Compare Structure Members Individually

One possible correction is to compare the structure members individually and to ensure that the difference between the floating-point values is within an acceptable range defined by `ESP`.

```
 #include <string.h>

typedef struct {
    int i;
    float f;
} myStruct;

extern void initialize_Struct(myStruct *);

#define ESP 0.00001

int func_cmp(myStruct *s1, myStruct *s2) {

/*Structure members are compared individually */
    return ((s1->i == s2->i) &&
            (fabsf(s1->f - s2->f) <= ESP));
}

void func(void) {
    myStruct s1, s2;
    initialize_Struct(&s1);
    initialize_Struct(&s2);
    (void)func_cmp(&s1, &s2);
}
```

## Check Information

**Group:** Rule 05. Floating Point (FLP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**
“Check for and Review Coding Standard Violations”

**External Websites**
FLP37-C

# CERT C: Rule ARR30-C

Do not form or use out-of-bounds pointers or array subscripts

## Description

### Rule Definition

*Do not form or use out-of-bounds pointers or array subscripts.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access out of bounds**.
- **Pointer access out of bounds**.
- **Array access with tainted index**.
- **Pointer dereference with tainted offset**.

### Extend Checker

A default Bug Finder analysis might not flag an **Array access out of bounds** issue when the input values are unknown and only a subset of inputs cause the issue. To check for the **Array access out of bounds** issue caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

A default Bug Finder analysis might not flag an **Array access with tainted index** or **Pointer dereference with tainted offset** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Array access out of bounds

#### Issue

**Array access out of bounds** occurs when an array index falls outside the range `[0...array_size-1]` during array access.

#### Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.

- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);    //Noncompliant
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

#### Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

### Pointer access out of bounds

#### Issue

**Pointer access out of bounds** occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

#### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i; //Noncompliant
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

#### Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

### Array access with tainted index

#### Issue

**Array access with tainted index** detects reading or writing to an array by using a tainted index that has not been validated.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.

- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

**Fix**

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

**Example - Use Index to Return Buffer Value**

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];   //Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The function does not check to see if `num` is inside the range of `tab`.

**Correction — Check Range Before Use**

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

### Pointer dereference with tainted offset

**Issue**

**Pointer dereference with tainted offset** detects pointer dereferencing, either reading or writing, using an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

#### Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

#### Example - Dereference Pointer Array

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset]; //Noncompliant
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of `offset`. Continue with the pointer dereferencing only if `offset` is inside the valid range.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

## Check Information

**Group:** Rule 06. Arrays (ARR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ARR30-C

# CERT C: Rule ARR32-C

Ensure size arguments for variable length arrays are in a valid range

## Description

### Rule Definition

*Ensure size arguments for variable length arrays are in a valid range.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Memory allocation with tainted size**.
- **Tainted size of variable length array**.

### Extend Checker

A default Bug Finder analysis might not flag a **Memory allocation with tainted size** or **Tainted size of variable length array** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Memory allocation with tainted size

#### Issue

**Memory allocation with tainted size** checks memory allocation functions, such as `calloc` or `malloc`, for size arguments from unsecured sources.

#### Risk

Uncontrolled memory allocation can cause your program to request too much system memory. This consequence can lead to a crash due to an out-of-memory condition, or assigning too many resources.

#### Fix

Before allocating memory, check the value of your arguments to check that they do not exceed the bounds.

#### Example - Allocate Memory Using Input Argument

```
#include<stdio.h>
#include <stdlib.h>

int* bug_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = (int*)malloc(size); //Noncompliant
    return p;
}
```

In this example, malloc allocates size amount of memory for the pointer p. size is an outside variable, so could be any size value. If the size is larger than the amount of memory you have available, your program could crash.

#### Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the malloc operation. This example checks to see if the size is positive and less than the maximum size.

```
#include<stdio.h>
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int* corrected_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = NULL;
    if (size>0 && size<SIZE128) {          /* Fix: Check entry range before use */
        p = (int*)malloc((unsigned int)size);
    }
    return p;
}
```

### Tainted size of variable length array

#### Issue

**Tainted size of variable length array** detects variable length arrays (VLA) whose size is from an unsecure source.

#### Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

#### Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

#### Example - Input Argument Used as Size of VLA

```
#include<stdio.h>
#inclule<stdlib.h>
#define LIM 40

long squaredSum(int size) {

    int tabvla[size]; //Noncompliant
    long res = 0;
    for (int i=0 ; i<LIM-1 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

**Correction — Check VLA Size**

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 10 and less than 100, before creating the VLA

```
#include <stdio.h>
#include <stdlib.h>
#define LIM 40

long squaredSum(int size) {
    long res = 0;
    if (size>0 && size<LIM){
        int tabvla[size];
        for (int i=0 ; i<size || i<LIM-1 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }else{
        res = -1;
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

## Check Information

**Group:** Rule 06. Arrays (ARR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ARR32-C

# CERT C: Rule ARR36-C

Do not subtract or compare two pointers that do not refer to the same array

## Description

### Rule Definition

*Do not subtract or compare two pointers that do not refer to the same array.*

### Polyspace Implementation

The rule checker checks for **Subtraction or comparison between pointers to different arrays**.

## Examples

### Subtraction or comparison between pointers to different arrays

#### Issue

**Subtraction or comparison between pointers to different arrays** occurs when you subtract or compare pointers that are null or that point to elements in different arrays. The relational operators for the comparison are >, <, >=, and <=.

#### Risk

When you subtract two pointers to elements in the same array, the result is the difference between the subscripts of the two array elements. Similarly, when you compare two pointers to array elements, the result is the positions of the pointers relative to each other. If the pointers are null or point to different arrays, a subtraction or comparison operation is undefined. If you use the subtraction result as a buffer index, it can cause a buffer overflow.

#### Fix

Before you subtract or use relational operators to compare pointers to array elements, check that they are non-null and that they point to the same array.

#### Example - Subtraction Between Pointers to Elements in Different Arrays

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int end;
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation is undefined unless array nums
    is adjacent to variable end in memory. */
    free_elements = &end - next_num_ptr;  //Noncompliant
    return free_elements;
}
```

In this example, the array `nums` is incrementally filled. Pointer subtraction is then used to determine how many free elements remain. Unless `end` points to a memory location one past the last element of `nums`, the subtraction operation is undefined.

**Correction — Subtract Pointers to the Same Array**

Subtract the pointer to the last element that was filled from the pointer to the last element in the array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation involves pointers to the same array. */
    free_elements = &(nums[SIZE20 - 1]) - next_num_ptr;

    return free_elements + 1;
}
```

## Check Information

**Group:** Rule 06. Arrays (ARR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ARR36-C

# CERT C: Rule ARR37-C

Do not add or subtract an integer to a pointer to a non-array object

## Description

### Rule Definition

*Do not add or subtract an integer to a pointer to a non-array object.*

### Polyspace Implementation

The rule checker checks for **Invalid assumptions about memory organization**.

## Examples

### Invalid assumptions about memory organization

#### Issue

**Invalid assumptions about memory organization** occurs when you compute the address of a variable in the stack by adding or subtracting from the address of another non-array variable.

#### Risk

When you compute the address of a variable in the stack by adding or subtracting from the address of another variable, you assume a certain memory organization. If your assumption is incorrect, accessing the computed address can be invalid.

#### Fix

Do not perform an access that relies on assumptions about memory organization.

If you want to justify this violation, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Reliance on Memory Organization

```
void func(void) {
    int var1 = 0x00000011, var2;
    *(&var1 + 1) = 0; //Noncompliant
}
```

In this example, the programmer relies on the assumption that `&var1 + 1` provides the address of `var2`. Therefore, an **Invalid assumptions about memory organization** appears on the `+` operation. In addition, a **Pointer access out of bounds** error also appears on the dereference.

#### Correction — Do Not Rely on Memory Organization

One possible correction is not perform direct computation on addresses to access separately declared variables.

### Check Information

**Group:** Rule 06. Arrays (ARR)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

ARR37-C

# CERT C: Rule ARR38-C

Guarantee that library functions do not form invalid pointers

## Description

### Rule Definition

*Guarantee that library functions do not form invalid pointers.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Mismatch between data length and size**.
- **Invalid use of standard library memory routine**.
- **Possible misuse of sizeof**.
- **Buffer overflow from incorrect string format specifier**.
- **Invalid use of standard library string routine**.
- **Destination buffer overflow in string manipulation**.
- **Destination buffer underflow in string manipulation**.

## Examples

### Mismatch between data length and size

#### Issue

**Mismatch between data length and size** looks for memory copying functions such as `memcpy`, `memset`, or `memmove`. If you do not control the length argument and data buffer argument properly, Bug Finder raises a defect.

#### Risk

If an attacker can manipulate the data buffer or length argument, the attacker can cause buffer overflow by making the actual data size smaller than the length.

This mismatch in length allows the attacker to copy memory past the data buffer to a new location. If the extra memory contains sensitive information, the attacker can now access that data.

This defect is similar to the SSL Heartbleed bug.

#### Fix

When copying or manipulating memory, compute the length argument directly from the data so that the sizes match.

#### Example - Copy Buffer of Data

```
#include <stdlib.h>
#include <string.h>
```

```
typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    memcpy(&(beta.data[num]), os->data + 2, length); //Noncompliant

    return(1);
}
```

This function copies the buffer `alpha` into a buffer `beta`. However, the `length` variable is not related to `data+2`.

#### Correction — Check Buffer Length

One possible correction is to check the length of your buffer against the maximum value minus 2. This check ensures that you have enough space to copy the data to the `beta` structure.

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    if (length<(os->max -2)) {
        memcpy(&(beta.data[num]), os->data + 2, length);
    }

    return(1);

}
```

### Invalid use of standard library memory routine

#### Issue

**Invalid use of standard library memory routine** occurs when a memory library function is called with invalid arguments. For instance, the memcpy function copies to an array that cannot accommodate the number of bytes copied.

#### Risk

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Use of Standard Library Memory Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%9s",str1);

  memcpy(str2,str1,6);  //Noncompliant
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string str2 is 5, but six characters of string str1 are copied into str2 using the memcpy function.

#### Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of str2 so that it accommodates the characters copied with the memcpy function.

```
#include <string.h>
#include <stdio.h>
```

```
char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%9s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

### Possible misuse of sizeof

#### Issue

**Possible misuse of sizeof** occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

#### Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

#### Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.

- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

#### Example - `sizeof` Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    { //Noncompliant
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

#### Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

### Buffer overflow from incorrect string format specifier

#### Issue

**Buffer overflow from incorrect string format specifier** occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

#### Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

#### Fix

Use a format specifier that is compatible with the memory buffer size.

#### Example - Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf); //Noncompliant
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

#### Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

### Invalid use of standard library string routine

#### Issue

**Invalid use of standard library string routine** occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source);
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Use of Standard Library String Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
{
 char *res;
 char gbuffer[5],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);  //Noncompliant
 /* Error: Size of text is less than gbuffer */
```

```
  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

#### Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Destination buffer overflow in string manipulation

#### Issue

**Destination buffer overflow in string manipulation** occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>
```

```
void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string); //Noncompliant
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

**Correction — Use `snprintf` Instead of `sprintf`**

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Destination buffer underflow in string manipulation

**Issue**

**Destination buffer underflow in string manipulation** occurs when certain string manipulation functions write to their destination buffer argument at a negative offset from the beginning of the buffer.

For instance, for the function `sprintf(char* buffer, const char* format)`, you obtain the `buffer` from an operation `buffer = (char*)arr; ... buffer += offset;`. `arr` is an array and `offset` is a negative value.

**Risk**

Buffer underflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer underflow also introduces the risk of code injection.

**Fix**

If the destination buffer argument results from pointer arithmetic, see if you are decrementing a pointer. Fix the pointer decrement by modifying either the original value before decrement or the decrement value.

**Example - Buffer Underflow in `sprintf` Use**

```
#include <stdio.h>
#define offset -2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);      //Noncompliant
}
```

In this example, `&buffer[offset]` is at a negative offset from the memory allocated to `buffer`.

#### Correction — Change Pointer Decrementer

One possible correction is to change the value of `offset`.

```
#include <stdio.h>
#define offset 2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

## Check Information

**Group:** Rule 06. Arrays (ARR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ARR38-C

# CERT C: Rule ARR39-C

Do not add or subtract a scaled integer to a pointer

## Description

### Rule Definition

*Do not add or subtract a scaled integer to a pointer.*

### Polyspace Implementation

The rule checker checks for **Incorrect pointer scaling**.

## Examples

### Incorrect pointer scaling

#### Issue

**Incorrect pointer scaling** occurs when Polyspace Bug Finder considers that you are ignoring the implicit scaling in pointer arithmetic.

For instance, the defect can occur in the following situations.

| Situation | Risk | Possible Fix |
|---|---|---|
| You use the `sizeof` operator in arithmetic operations on a pointer. | The `sizeof` operator returns the size of a data type in number of bytes.<br>Pointer arithmetic is already implicitly scaled by the size of the data type of the pointed variable. Therefore, the use of `sizeof` in pointer arithmetic produces unintended results. | Do not use `sizeof` operator in pointer arithmetic. |
| You perform arithmetic operations on a pointer, and then apply a cast. | Pointer arithmetic is implicitly scaled. If you do not consider this implicit scaling, casting the result of a pointer arithmetic produces unintended results. | Apply the cast before the pointer arithmetic. |

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Example — Use of sizeof Operator

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2*(sizeof(int)));//Noncompliant //Noncompliant
}
```

In this example, the operation `2*(sizeof(int))` returns twice the size of an `int` variable in bytes. However, because pointer arithmetic is implicitly scaled, the number of bytes by which `ptr` is offset is `2*(sizeof(int))*(sizeof(int))`.

In this example, the incorrect scaling shifts `ptr` outside the bounds of the array. Therefore, a **Pointer access out of bounds** error appears on the * operation.

### Correction — Remove sizeof Operator

One possible correction is to remove the `sizeof` operator.

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2);
}
```

### Example — Cast Following Pointer Arithmetic

```
int func(void) {
    int x = 0;
    char r = *(char *)(&x + 1); //Noncompliant
    return r;
}
```

In this example, the operation `&x + 1` offsets `&x` by `sizeof(int)`. Following the operation, the resulting pointer points outside the allowed buffer. When you dereference the pointer, a **Pointer access out of bounds** error appears on the * operation.

### Correction — Apply Cast Before Pointer Arithmetic

If you want to access the second byte of `x`, first cast `&x` to a `char*` pointer and then perform the pointer arithmetic. The resulting pointer is offset by `sizeof(char)` bytes and still points within the allowed buffer, whose size is `sizeof(int)` bytes.

```
int func(void) {
    int x = 0;
    char r = *((char *)(&x )+ 1);
    return r;
}
```

**Example — Use of sizeof in Function Arguments**

```
#include <stddef.h>
#include <stdlib.h>
#include <wchar.h>
enum { WCHAR_BUF = 128 };
FILE* pFile;
//...
void func2_ko (void)
{
    wchar_t error_msg[WCHAR_BUF];
    wcscpy (error_msg, L"Error: ");
    fgetws (error_msg + wcslen (error_msg)   //Noncompliant
           * sizeof (wchar_t), WCHAR_BUF - 7, pFile);
}
```

In this example, an error message is read from the file pointer pFile stream and copied to error_msg after an offset. The intended offset here is wcslen(error_msg), which is already implicitly scaled when it is added to the wchar pointer error_msg. Because the offset is then explicitly scaled again by using sizeof, Polyspace flags the incorrect scaling.

**Correction — Remove sizeof Operator**

One possible correction is to remove the sizeof operator.

```
#include <stddef.h>
#include <stdlib.h>
#include <wchar.h>
enum { WCHAR_BUF = 128 };
const wchar_t ERROR_PREFIX[8] = L"Error: ";
FILE* pFile;
//...

void func2_ok (void)
{
  const size_t prefix_len = wcslen (ERROR_PREFIX);
  wchar_t error_msg[WCHAR_BUF];
  wcscpy (error_msg, ERROR_PREFIX);
  fgetws (error_msg + prefix_len, WCHAR_BUF - prefix_len, pFile);  //Compliant
  /* ... */
}
```

**Example — Implicitly Scaled Offset When Calling memset**

```
#include <string.h>
#include <stdlib.h>
#include <stddef.h>

#define bigNum unsigned long long

struct Collection {
    bigNum bn_A;
    bigNum bn_B;
    bigNum bn_C;
    int ci_1;
    int ci_2;
};

void foo(void) {
    size_t offset = offsetof(struct Collection, bn_B);
    struct Collection *s = (struct Collection *)malloc(sizeof(struct Collection));
    if (s == NULL) {
        /* Handle malloc() error */
    }

    memset(s + offset, 0, sizeof(struct Collection) - offset); //Noncompliant
    /* ... */
    free(s);
    s = NULL;
}
```

In this example, `offset` is calculated by calling `offsetof`, and then added to `s`. The variable `offset` is the byte offset of `bn_B` in `struct Collection`. When setting memory by using `memset`, instead of offsetting the location by bytes, `offset` is implicitly scaled by the size. The implicit scaling might cause unexpected results. Polyspace raises a violation.

#### Correction — Calculate Offset by Using `unsigned char*`

The violation is caused by the fact that `offset` is scaled by the type of `s`. Avoid the violation by declaring `s` as `unsigned char*`, which is scaled by a factor of one.

```
#include <string.h>
#include <stdlib.h>
#include <stddef.h>

#define bigNum unsigned long long

struct Collection {
    bigNum bn_A;
    bigNum bn_B;
    bigNum bn_C;
    int ci_1;
    int ci_2;
};

void foo(void) {
    size_t offset = offsetof(struct Collection, bn_B);
    unsigned char *s = (unsigned char *)malloc(sizeof(struct Collection));
    if (s == NULL) {
        /* Handle malloc() error */
    }

    memset(s + offset, 0, sizeof(struct Collection) - offset); //Compliant
    /* ... */
    free(s);
    s = NULL;
}
```

## Check Information

**Group:** Rule 06. Arrays (ARR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

ARR39-C

# CERT C: Rule STR30-C

Do not attempt to modify string literals

## Description

### Rule Definition

*Do not attempt to modify string literals.*

### Polyspace Implementation

The rule checker checks for **Writing to const qualified object**.

## Examples

### Writing to const qualified object

#### Issue

**Writing to `const` qualified object** occurs when you do one of the following:

- Use a `const`-qualified object as the destination of an assignment.
- Pass a `const`-qualified object to a function that modifies the argument.

For instance, the defect can occur in the following situations:

- You pass a `const`-qualified object as first argument of one of the following functions:
  - `mkstemp`
  - `mkostemp`
  - `mkostemps`
  - `mkdtemp`
- You pass a `const`-qualified object as the destination argument of one of the following functions:
  - `strcpy`
  - `strncpy`
  - `strcat`
  - `memset`
- You perform a write operation on a `const`-qualified object.

#### Risk

The risk depends upon the modifications made to the `const`-qualified object.

| Situation | Risk |
|---|---|
| Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on. | These functions replace the last six characters of their first argument with a string. Therefore, they expect a modifiable `char` array as their first argument. |
| Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on. | These functions modify their destination argument. Therefore, they expect a modifiable `char` array as their destination argument. |
| Writing to the object | The `const` qualifier implies an agreement that the value of the object will not be modified. By writing to a `const`-qualified object, you break the agreement. The result of the operation is undefined. |

#### Fix

The fix depends on the modification made to the `const`-qualified object.

| Situation | Fix |
|---|---|
| Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on. | Pass a non-`const` object as first argument of the function. |
| Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on. | Pass a non-`const` object as destination argument of the function. |
| Writing to the object | Perform the write operation on a non-`const` object. |

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Writing to `const`-Qualified Object

```
#include <string.h>

const char* buffer = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string); //Noncompliant
}
```

In this example, because `buffer` is `const`-qualified, `strchr(buffer,'X')` returns a `const`-qualified `char*` pointer. When this `char*` pointer is used as the destination argument of `strcpy`, a **Writing to const qualified object** error appears.

#### Correction — Copy `const`-Qualified Object to Non-`const` Object

One possible correction is to assign the constant string to a non-`const` object and use the non-`const` object as destination argument of `strchr`.

```
#include <string.h>

char buffer[] = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

## Check Information

**Group:** Rule 07. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

STR30-C

# CERT C: Rule STR31-C

Guarantee that storage for strings has sufficient space for character data and the null terminator

## Description

### Rule Definition

*Guarantee that storage for strings has sufficient space for character data and the null terminator.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Use of dangerous standard function**.
- **Missing null in string array**.
- **Buffer overflow from incorrect string format specifier**.
- **Destination buffer overflow in string manipulation**.
- **Insufficient destination buffer size**.

## Examples

### Use of dangerous standard function

#### Issue

The **Use of dangerous standard function** check highlights uses of functions that are inherently dangerous or potentially dangerous given certain circumstances. The following table lists possibly dangerous functions, the risks of using each function, and what function to use instead.

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `gets` | Inherently dangerous — You cannot control the length of input from the console. | `fgets` |
| `cin` | Inherently dangerous — You cannot control the length of input from the console. | Avoid or prefaces calls to `cin` with `cin.width`. |
| `strcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `strncpy` |
| `stpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `stpncpy` |
| `lstrcpy` or `StrCpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy` |

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `strcat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `strncat`, `strlcat`, or `strcat_s` |
| `lstrcat` or `StrCat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat` |
| `wcpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcpncpy` |
| `wcscat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `wcsncat`, `wcslcat`, or `wcncat_s` |
| `wcscpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcsncpy` |
| `sprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `snprintf` |
| `vsprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `vsnprintf` |

#### Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Using `sprintf`

```
#include <stdio.h>
#include <string.h>
#include <iostream>
```

```
#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1) //Noncompliant
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

#### Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

### Missing null in string array

#### Issue

**Missing null in string array** occurs when a string does not have enough space to terminate with a null character `'\0'`.

This defect applies only for projects in C.

#### Risk

A buffer overflow can occur if you copy a string to an array without assuming the implicit null terminator.

#### Fix

If you initialize a character array with a literal, avoid specifying the array bounds.

```
char three[]  = "THREE";
```

The compiler automatically allocates space for a null terminator. In the preceding example, the compiler allocates sufficient space for five characters and a null terminator.

If the issue occurs after initialization, you might have to increase the size of the array by one to account for the null terminator.

In certain circumstances, you might want to initialize the character array with a sequence of characters instead of a string. In this situation, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Array size is too small

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE"; //Noncompliant
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

#### Correction — Increase Array Size

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

#### Correction — Change Initialization Method

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

### Buffer overflow from incorrect string format specifier

#### Issue

**Buffer overflow from incorrect string format specifier** occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

#### Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

#### Fix

Use a format specifier that is compatible with the memory buffer size.

#### Example - Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf); //Noncompliant
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

#### Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

### Destination buffer overflow in string manipulation

#### Issue

**Destination buffer overflow in string manipulation** occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string); //Noncompliant
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

#### Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Insufficient destination buffer size

#### Issue

**Insufficient destination buffer size** occurs when the destination buffer in a `strcpy` operation cannot accommodate the source buffer and a null terminator. This issue is reported if the size of the source buffer is unknown. Consider this code:

```
int main (int argc, char *argv[])
{
  const char *const input = ((argc && argv[0]) ? argv[0] : "");
  char str[100];
  strcpy(input, str); // Noncompliant
}
```

In this case, the size of the source buffer `input` is unknown. The size of the destination buffer `str` might be smaller than the value `(strlen(input)+1)`. Polyspace reports a violation on the `strcpy` operation.

#### Risk

Using a destination buffer of insufficient size might allow an attacker to cause a buffer overflow. In the preceding code example, if `argv[0]` contains 100 or more characters, the `strcpy` operation results in a buffer overflow.

#### Fix

Before calling the function `strcpy()`, allocate sufficient memory dynamically. For instance, use the function `strlen()` to determine the size of the source buffer and then allocate the destination buffer so that its size is greater than the value `strlen(source) + 1`.

#### Example — Destination Buffer Too Small

In this example, the size of the `source` buffer is unknown, while the size of the `destination` buffer is fixed at `128`. The size of the `destination` buffer might not be sufficient to accommodate the characters from the `source` buffer and terminate the buffer with a null. Polyspace reports a violation of the rule.

```
#include <string.h>

int main(int argc, char *argv[]) {
  const char *const source = (argc && argv[0]) ? argv[0] : "";
  char destination[128];
  strcpy(source, destination);//Noncompliant

  return 0;
}
```

#### Correction — Allocate Sufficient Memory for Destination Buffer

This violation is resolved by allocating sufficient memory for the `destination` buffer. For instance, use the function `strlen()` to calculate the size of the `source` buffer and allocate sufficient memory for the `destination` buffer so that it can accommodate all characters from the `source` buffer and the null terminator (`'\0'` ).

```
#include <string.h>

int main(int argc, char *argv[]) {
  const char *const source = (argc && argv[0]) ? argv[0] : "";
  char* destination = (char *)malloc(strlen(source)+ 1);
  if(destination!=NULL){
      strcpy(source, destination);//Compliant
  }else{
      /*Handle Error*/
  }
  //...
  free(destination);
  return 0;
}
```

## Check Information

**Group:** Rule 07. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

STR31-C

# CERT C: Rule STR32-C

Do not pass a non-null-terminated character sequence to a library function that expects a string

## Description

### Rule Definition

*Do not pass a non-null-terminated character sequence to a library function that expects a string.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Invalid use of standard library string routine**.
- **Tainted NULL or non-null-terminated string**.

### Extend Checker

A default Bug Finder analysis might not flag a **Tainted NULL or non-null-terminated string** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Invalid use of standard library string routine

#### Issue

**Invalid use of standard library string routine** occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);  //Noncompliant
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

#### Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Tainted NULL or non-null-terminated string

#### Issue

**Tainted NULL or non-null-terminated string** looks for strings from unsecure sources that are being used in string manipulation routines that implicitly dereference the string buffer. For example, `strcpy` or `sprintf`.

**Tainted NULL or non-null-terminated string** raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

**Note** If you reference a string using the form `ptr[i]`, `*ptr`, or pointer arithmetic, Bug Finder raises a **Use of tainted pointer** defect instead. The **Tainted NULL or non-null-terminated string** defect is raised only when the pointer is used as a string.

#### Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

#### Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

#### Example — Getting String from Input

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1)); //Noncompliant
    print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

#### Correction — Validate the Data

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
```

```
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // TAINTED_STRING only flagged here //Noncompliant
        // - string is not null
        // - string has a positive and limited size
        // - TAINTED_STRING on strlen used as a firewall
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

**Correction — Validate the Data**

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }
  return sensorValue;
}
```

## Check Information

**Group:** Rule 07. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

STR32-C

# CERT C: Rule STR34-C

Cast characters to unsigned char before converting to larger integer sizes

## Description

### Rule Definition

*Cast characters to unsigned char before converting to larger integer sizes.*

### Polyspace Implementation

The rule checker checks for **Misuse of sign-extended character value**.

## Examples

### Misuse of sign-extended character value

#### Issue

**Misuse of sign-extended character value** occurs when you convert a signed or plain `char` data type to a wider integer data type with sign extension. You then use the resulting sign-extended value as array index, for comparison with EOF or as argument to a character-handling function.

#### Risk

*Comparison with EOF*: Suppose, your compiler implements the plain `char` type as signed. In this implementation, the character with the decimal form of 255 (–1 in two's complement form) is stored as a signed value. When you convert a `char` variable to the wider data type `int` for instance, the sign bit is preserved (sign extension). This sign extension results in the character with the decimal form 255 being converted to the integer –1, which cannot be distinguished from EOF.

*Use as array index*: By similar reasoning, you cannot use sign-extended plain `char` variables as array index. If the sign bit is preserved, the conversion from `char` to `int` can result in negative integers. You must use positive integer values for array index.

*Argument to character-handling function*: By similar reasoning, you cannot use sign-extended plain `char` variables as arguments to character-handling functions declared in `ctype.h`, for instance, `isalpha()` or `isdigit()`. According to the C11 standard (Section 7.4), if you supply an integer argument that cannot be represented as `unsigned char` or EOF, the resulting behavior is undefined.

#### Fix

Before conversion to a wider integer data type, cast the signed or plain `char` value explicitly to `unsigned char`.

#### Example - Sign-Extended Character Value Compared with EOF

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];
```

```
static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = *buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {  //Noncompliant
        /* Handle error */
        fatal_error();
    }
}
```

In this example, the function `parser` can traverse a string input `buf`. If a character in the string has the decimal form 255, when converted to the `int` variable `c`, its value becomes –1, which is indistinguishable from `EOF`. The later comparison with `EOF` can lead to a false positive.

#### Correction — Cast to `unsigned char` Before Conversion

One possible correction is to cast the plain `char` value to `unsigned char` before conversion to the wider `int` type.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = (unsigned char)*buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

## Check Information

**Group:** Rule 07. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**

“Check for and Review Coding Standard Violations”

**External Websites**

STR34-C

# CERT C: Rule STR37-C

Arguments to character-handling functions must be representable as an unsigned char

## Description

### Rule Definition

*Arguments to character-handling functions must be representable as an unsigned char.*

### Polyspace Implementation

The rule checker checks for **Invalid arguments to character-handling functions**.

## Examples

### Invalid arguments to character-handling functions

#### Issue

**Invalid arguments to character-handling functions** occurs when you use a signed or plain `char` variable with a negative value as argument to a character-handling function declared in `ctype.h`, for instance, `isalpha()` or `isdigit()`.

#### Risk

You cannot use plain `char` variables as arguments to these character-handling functions. On certain platforms, plain `char` variables can have negative values that cannot be represented as `unsigned char` or EOF, resulting in undefined behavior.

#### Fix

To avoid unexpected results, explicitly cast plain `char` variables to unsigned `char` before passing to character-handling functions.

## Check Information

**Group:** Rule 07. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

STR37-C

# CERT C: Rule STR38-C

Do not confuse narrow and wide character strings and functions

## Description

### Rule Definition

*Do not confuse narrow and wide character strings and functions.*

### Polyspace Implementation

The rule checker checks for **Misuse of narrow or wide character string**.

## Examples

### Misuse of narrow or wide character string

#### Issue

**Misuse of narrow or wide character string** occurs when you pass a narrow character string to a wide string function, or a wide character string to a narrow string function.

**Misuse of narrow or wide character string** raises no defect on operating systems where narrow and wide character strings have the same size.

#### Risk

Using a narrow character string with a wide string function, or vice versa, can result in unexpected or undefined behavior.

If you pass a wide character string to a narrow string function, you can encounter these issues:

- Data truncation. If the string contains null bytes, a copy operation using `strncpy()` can terminate early.
- Incorrect string length. `strlen()` returns the number of characters of a string up to the first null byte. A wide string can have additional characters after its first null byte.

If you pass a narrow character string to a wide string function, you can encounter this issue:

- Buffer overflow. In a copy operation using `wcsncpy()`, the destination string might have insufficient memory to store the result of the copy.

#### Fix

Use the narrow string functions with narrow character strings. Use the wide string functions with wide character strings.

#### Example - Passing Wide Character Strings to `strncpy()`

```
#include <string.h>
#include <wchar.h>

void func(void)
```

```
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    strncpy(wide_str2, wide_str1, 10); //Noncompliant
}
```

In this example, `strncpy()` copies 10 wide characters from `wide_strt1` to `wide_str2`. If `wide_str1` contains null bytes, the copy operation can end prematurely and truncate the wide character string.

#### Correction — Use `wcsncpy()` to Copy Wide Character Strings

One possible correction is to use `wcsncpy()` to copy `wide_str1` to `wide_str2`.

```
#include <string.h>
#include <wchar.h>

void func(void)
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    wcsncpy(wide_str2, wide_str1, 10);
}
```

## Check Information

**Group:** Rule 07. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

STR38-C

# CERT C: Rule MEM30-C

Do not access freed memory

## Description

### Rule Definition

*Do not access freed memory.*

### Polyspace Implementation

This checker checks for:

- **Accessing previously freed pointer**
- **Freeing previously freed pointer**

## Examples

### Accessing previously freed pointer

#### Issue

**Accessing previously freed pointer** occurs when you attempt to access a block of memory after freeing the block by using the `free` function.

#### Risk

When a pointer is allocated dynamic memory by using `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation and the pointer becomes a dangling pointer. Attempting to access this block of memory by dereferencing the dangling pointer can result in unpredictable behavior or a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. Determine if you intended to free the memory later or allocate another memory block to the pointer before access.

As a best practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

#### Example — Accessing Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift; //Noncompliant
    /* Defect: Reading a freed pointer */

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing `pi` after the `free` statement is not valid.

**Correction — Free Pointer After Last Use**

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

### Freeing previously freed pointer

**Issue**

**Freeing previously freed pointer** occurs when you attempt to free the memory allocated to a pointer after already freeing the pointer by using the `free` function.

**Risk**

Attempting to free the memory associated with a previously freed pointer might corrupt the memory management of the program and cause a memory leak. This defect might allow an attacker to access the memory and execute arbitrary code.

**Fix**

To avoid this defect, assign pointers to NULL after freeing them. Check the pointers for NULL value before attempting to access the memory associated with the pointer. In this way, you are protected against accessing a freed block.

**Example — Freeing Previously Freed Pointer**

```
#include <stdlib.h>
#include <stdio.h>
int getStatus();
void double_deallocation(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == 0) return;

    *pi = 2;
    /*...*/
    if(getStatus()==1)
    {
        /*...*/
        free(pi);
    }
    free(pi); //Noncompliant //Noncompliant
}
```

The second `free` statement attempts to release the block of memory that `pi` refers to, but the pointer `pi` might already be freed in the `if` block of code. This second `free` statement might cause a memory leak and security vulnerabilities in the code. Polyspace flags the second `free` statement.

**Correction — Check Pointers Before Calling `free`**

One possible correction is to assign freed pointers to NULL and to check pointers for NULL before freeing them.

```
#include <stdlib.h>
#include <stdio.h>
int getStatus();
void double_deallocation(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == 0) return;

    *pi = 2;
    /*...*/
    if(getStatus()==1)
    {
        /*...*/
        if(pi!=NULL)
        {
            free(pi);
            pi= NULL;
        }
    }
    /*...*/
    if(pi!=NULL)
    {
        free(pi);
        pi= NULL;
    } //Compliant
}
```

In this case, the memory allocated to pointer `pi` is freed only if it is not already freed.

**Example – Freeing Pointer Previously Reallocated With Possibly Zero Size**

```
#include <stdlib.h>

void reshape(char *buf, size_t size) {
  char *reshaped_buf = (char *)realloc(buf, size);
  if (reshaped_buf == NULL) {
    free(buf); //Noncompliant
  }
}
```

In this example, the argument `size` of the `reshape()` function can be zero and result in a zero-size reallocation with `realloc()`. In some implementations such as the GNU library, zero-size reallocations free the memory leading to a double free defect.

#### Correction – Guard Against Zero-Size Reallocations

One possible correction is to check size argument of `realloc()` for zero values before use. If the size argument is zero, you can simply free the memory instead of reallocating it.

```
#include <stdlib.h>

void reshape(char *buf, size_t size) {
  if (size != 0) {
    char *reshaped_buf = (char *)realloc(buf, size);
    if (reshaped_buf == NULL) {
      free(buf);
    }
  }
  else {
    free(buf);
  }

}
```

## Check Information

**Group:** Rule 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

MEM30-C

# CERT C: Rule MEM31-C

Free dynamically allocated memory when no longer needed

## Description

### Rule Definition

*Free dynamically allocated memory when no longer needed.*

### Polyspace Implementation

The rule checker checks for **Memory leak**.

## Examples

### Memory leak

#### Issue

**Memory leak** occurs when you do not free a block of memory allocated through `malloc`, `calloc`, `realloc`, or `new`. If the memory is allocated in a function, the defect does not occur if:

- Within the function, you free the memory using `free` or `delete`.
- The function returns the pointer assigned by `malloc`, `calloc`, `realloc`, or `new`.
- The function stores the pointer in a global variable or in a parameter.

#### Risk

Dynamic memory allocation functions such as `malloc` allocate memory on the heap. If you do not release the memory after use, you reduce the amount of memory available for another allocation. On embedded systems with limited memory, you might end up exhausting available heap memory even during program execution.

#### Fix

Determine the scope where the dynamically allocated memory is accessed. Free the memory block at the end of this scope.

To free a block of memory, use the `free` function on the pointer that was used during memory allocation. For instance:

```
ptr = (int*)malloc(sizeof(int));
//...
free(ptr);
```

It is a good practice to allocate and free memory in the same module at the same level of abstraction. For instance, in this example, `func` allocates and frees memory at the same level but `func2` does not.

```
void func() {
  ptr = (int*)malloc(sizeof(int));
  {
    //...
  }
  free(ptr);
}

void func2() {
  {
   ptr = (int*)malloc(sizeof(int));
   //...
  }
  free(ptr);
}
```

See CERT-C Rule MEM00-C.

**Example - Dynamic Memory Not Released Before End of Function**

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
} //Noncompliant
```

In this example, `pi` is dynamically allocated by `malloc`. The function `assign_memory` does not free the memory, nor does it return `pi`.

**Correction — Free Memory**

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
    free(pi);
}
```

#### Correction — Return Pointer from Dynamic Allocation

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

```
#include<stdlib.h>
#include<stdio.h>

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) 
        {
            printf("Memory allocation failed");
            return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

## Check Information

**Group:** Rule 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MEM31-C

# CERT C: Rule MEM33-C

Allocate and copy structures containing a flexible array member dynamically

## Description

### Rule Definition

*Allocate and copy structures containing a flexible array member dynamically.*

### Polyspace Implementation

The rule checker checks for **Misuse of structure with flexible array member**.

## Examples

### Misuse of structure with flexible array member

#### Issue

**Misuse of structure with flexible array member** occurs when:

- You define an object with a flexible array member of unknown size at compilation time.
- You make an assignment between structures with a flexible array member without using `memcpy()` or a similar function.
- You use a structure with a flexible array member as an argument to a function and pass the argument by value.
- Your function returns a structure with a flexible array member.

A flexible array member has no array size specified and is the last element of a structure with at least two named members.

#### Risk

If the size of the flexible array member is not defined, it is ignored when allocating memory for the containing structure. Accessing such a structure has undefined behavior.

#### Fix

- Use `malloc()` or a similar function to allocate memory for a structure with a flexible array member.
- Use `memcpy()` or a similar function to copy a structure with a flexible array member.
- Pass a structure with a flexible array member as a function argument by pointer.

#### Example - Structure Passed By Value to Function

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
```

```
struct example_struct
{
    size_t num;
    int data[];
};

extern void arg_by_value(struct example_struct s);

void func(void)
{
    struct example_struct *flex_struct;
    size_t i;
    size_t array_size = 4;
    /* Dynamically allocate memory for the struct */
    flex_struct = (struct example_struct *)
        malloc(sizeof(struct example_struct) + sizeof(int) * array_size);
    if (flex_struct == NULL)
    {
        /* Handle error */
    }
    /* Initialize structure */
    flex_struct->num = array_size;
    for (i = 0; i < array_size; ++i)
    {
        flex_struct->data[i] = 0;
    }
    /* Handle structure */

    /* Argument passed by value. 'data' not
    copied to passed value. */
    arg_by_value(*flex_struct);  //Noncompliant

    /* Free dynamically allocated memory */
    free(flex_struct);
}
```

In this example, `flex_struct` is passed by value as an argument to `arg_by_value`. As a result, the flexible array member data is not copied to the passed argument.

**Correction — Pass Structure by Pointer to Function**

To ensure that all the members of the structure are copied to the passed argument, pass `flex_struct` to `arg_by_pointer` by pointer.

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>


struct example_struct
{
    size_t num;
    int data[];
};

extern void arg_by_pointer(struct example_struct *s);

void func(void)
{
    struct example_struct *flex_struct;
    size_t i;
    size_t array_size = 4;
    /* Dynamically allocate memory for the struct */
    flex_struct = (struct example_struct *)
        malloc(sizeof(struct example_struct) + sizeof(int) * array_size);
    if (flex_struct == NULL)
    {
        /* Handler error */
    }
    /* Initialize structure */
    flex_struct->num = array_size;
    for (i = 0; i < array_size; ++i)
    {
        flex_struct->data[i] = 0;
    }
    /* Handle structure */

    /* Structure passed by pointer */
    arg_by_pointer(flex_struct);

    /* Free dynamically allocated memory */
    free(flex_struct);
}
```

## Check Information

**Group:** Rule 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MEM33-C

# CERT C: Rule MEM34-C

Only free memory allocated dynamically

## Description

### Rule Definition

*Only free memory allocated dynamically.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Invalid free of pointer**
- **Invalid reallocation of pointer**

## Examples

### Invalid free of pointer

#### Issue

**Invalid free of pointer** occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

#### Risk

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function on a different pointer.

#### Fix

In most cases, you can fix the issue by removing the `free` statement. If a pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

#### Example - Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
```

```
     *(p+i)=1;

  free(p);    //Noncompliant
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer p is deallocated using the free function. However, p points to a memory location that was not dynamically allocated.

#### Correction — Remove Pointer Deallocation

If the number of elements of the array p is known at compile time, one possible correction is to remove the deallocation of the pointer p.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

#### Correction — Introduce Pointer Allocation

If the number of elements of the array p is not known at compile time, one possible correction is to dynamically allocate memory to the array p.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

### Invalid reallocation of pointer

#### Issue

**Invalid reallocation of pointer** occurs when a block of memory reallocated using the realloc function was not previously allocated using malloc or calloc.

#### Risk

Reallocating a block of memory that was not allocated dynamically allocated can result in undefined behavior.

The issue can highlight coding errors. For instance, you perhaps wanted to use the realloc function on a different pointer.

**Fix**

If you want to reallocate a block of memory, make sure that it was dynamically allocated in the first place.

**Example – Reallocation of Memory Allocated Statically**

```
#include <stdlib.h>

#define SIZE 256

void reshape(int isSpaceAvailable) {
  char buf[SIZE];
  char *newBuf;
  newBuf = (char *)realloc(buf, 2 * SIZE); //Noncompliant

  if (newBuf == NULL) {
    /* Handle error */
  }
}
```

In this example, the buffer `buf` is not allocated dynamically. Therefore, the reallocation of `buf` results in undefined behavior.

**Example – Reallocation of Memory Allocated Statically**

Make sure that a buffer that you reallocate was previous allocated memory dynamically.

```
#include <stdlib.h>

#define SIZE 256

void reshape(void){
  char *buf;
  char *newBuf;
  buf = (char*)malloc(SIZE *sizeof(char));
  newBuf = (char *)realloc(buf, 2 * SIZE);

  if (newBuf == NULL) {
    /* Handle error */
  }
  free(newBuf);
}
```

## Check Information

**Group:** Rule 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

MEM34-C

# CERT C: Rule MEM35-C

Allocate sufficient memory for an object

## Description

### Rule Definition

*Allocate sufficient memory for an object.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Pointer access out of bounds**.
- **Memory allocation with tainted size**.
- **Insufficient memory allocation**

### Extend Checker

A default Bug Finder analysis might not flag a **Memory allocation with tainted size** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Pointer access out of bounds

#### Issue

**Pointer access out of bounds** occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

#### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i; //Noncompliant
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

#### Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

### Memory allocation with tainted size

#### Issue

**Memory allocation with tainted size** checks memory allocation functions, such as `calloc` or `malloc`, for size arguments from unsecured sources.

#### Risk

Uncontrolled memory allocation can cause your program to request too much system memory. This consequence can lead to a crash due to an out-of-memory condition, or assigning too many resources.

#### Fix

Before allocating memory, check the value of your arguments to check that they do not exceed the bounds.

#### Example — Allocate Memory Using Input From User

```
#include<stdio.h>
#include <stdlib.h>

int* bug_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = (int*)malloc(size); //Noncompliant
    return p;
}
```

In this example, `malloc` allocates `size` bytes of memory for the pointer `p`. The variable `size` comes from the user of the program. Its value is not checked, and it could be larger than the amount of available memory. If `size` is larger than the number of available bytes, your program could crash.

#### Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the `malloc` operation. This example checks to see if `size` is positive and less than the maximum size.

```
#include<stdio.h>
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int* corrected_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = NULL;
    if (size>0 && size<SIZE128) {          /* Fix: Check entry range before use */
        p = (int*)malloc((unsigned int)size);
    }
    return p;
}
```

### Insufficient memory allocation

#### Issue

**Insufficient memory allocation** occurs when the allocated memory is not sufficient to hold the target object. Consider this code:

```
typedef struct S {
    int a;
    int b;
    int c;
}S;

void foo(){
    S* a;
    a = (S*)malloc(sizeof(S*)*2);//Noncompliant
}
void bar(){
    S* a;
    a = (S*)malloc(sizeof(S*)*3); //Compliant
}
```

Objects of type S must accommodate three integers, which typically requires 12 bytes of memory. In foo() the allocated memory size for the S type object is eight bytes. Because the allocated memory is insufficient to hold the target object, Polyspace reports a violation of this rule. In bar(), the allocated memory is 12 bytes, which is sufficient. Polyspace does not report a violation.

#### Risk

Insufficient memory allocation results in buffer overflow and unexpected termination of the program.

#### Fix

When allocating memory blocks, allocate sufficient memory.

#### Example — Allocate Memory for Objects by Calling malloc()

In this example, the function foo() allocates memory for pointers a and b:

- When allocating memory for a, the argument of sizeof() is S*, which is a pointer. In a 64 bit system, all pointer types are 8 bytes in size. The malloc() statement allocates 24 bytes, which is sufficient to accommodate an object of S type. Because the allocated memory can hold the a object, Polyspace does not report a violation. This statement causes a Wrong type used in sizeof defect.

  From the context of the malloc() statement, the object a is intended to be a three element array of S type objects. Clearly, the malloc() statement does not allocate sufficient memory for the entire array. In the for loop, when you access the elements of the array, the insufficient memory causes a violation of this rule. This way, the Wrong type used in sizeof defect in the malloc() statement causes a violation of this rule when you use a as an array. If you use a as an object instead of an array, this rule is not violated.
- When allocating memory for b, the malloc() statement allocates enough memory to accommodate two integers. Because objects of type S require enough memory to accommodate three integers, the allocated memory is insufficient. Polyspace reports a violation of this rule.

```
#include <stdlib.h>
```

```
typedef struct S
{
  int a;
  int b;
  int c;
} S;

void setS (S * a, int x, int y, int z)
{
  a->a = x;            //Noncompliant: pointer access out of bounds
  a->b = y;
  a->c = z;
}

void foo ()
{
  S *a, *b;
  a = (S *) malloc (3 * sizeof (S *));    //Compliant but causes PTR_SIZEOF_MISMATCH defect/
  setS(a,22,32,45); //Compliant
  for (int i = 1; i < 3; ++i)
    {
      setS (a + i, i, i * i, i * i * i);

    }
  b = (S *) malloc (2 * sizeof (int));    //Noncompliant

}
```

### Correction — Allocate Sufficient Memory

To fix this issue, allocate sufficient memory. The best practice is to use the `sizeof()` function with correct argument type and a correct multiplier to calculate how much memory is required.

```
#include <stdlib.h>


typedef struct S
{
  int a;
  int b;
  int c;
} S;

void setS (S * a, int x, int y, int z)
{
  a->a = x;            //Compliant: pointer access out of bounds
  a->b = y;
  a->c = z;
}

void foo ()
{
  S *a, *b;
  a = (S *) malloc (3 * sizeof (S));    //No violations of PTR_SIZEOF_MISMATCH*/
  setS(a,22,32,45); //Compliant
  for (int i = 1; i < 3; ++i)
    {
      setS (a + i, i, i * i, i * i * i);

    }
  b = (S *) malloc (1 * sizeof (S));    //Compliant
  //...
  free(a);
  free (b);
}
```

### Check Information

**Group:** Rule 08. Memory Management (MEM)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

MEM35-C

# CERT C: Rule MEM36-C

Do not modify the alignment of objects by calling realloc()

## Description

### Rule Definition

*Do not modify the alignment of objects by calling realloc().*

### Polyspace Implementation

The rule checker checks for **Alignment changed after memory reallocation**.

## Examples

### Alignment changed after memory reallocation

#### Issue

**Alignment changed after memory reallocation** occurs when you use `realloc()` to modify the size of objects with strict memory alignment requirements.

#### Risk

The pointer returned by `realloc()` can be suitably assigned to objects with less strict alignment requirements. A misaligned memory allocation can lead to buffer underflow or overflow, an illegally dereferenced pointer, or access to arbitrary memory locations. In processors that support misaligned memory, the allocation impacts the performance of the system.

#### Fix

To reallocate memory:

1. Resize the memory block.
   - In Windows, use `_aligned_realloc()` with the alignment argument used in `_aligned_malloc()` to allocate the original memory block.
   - In UNIX/Linux, use the same function with the same alignment argument used to allocate the original memory block.
2. Copy the original content to the new memory block.
3. Free the original memory block.

**Note** This fix has implementation-defined behavior. The implementation might not support the requested memory alignment and can have additional constraints for the size of the new memory.

#### Example - Memory Reallocated Without Preserving the Original Alignment

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define SIZE1024 1024

void func(void)
{
    size_t resize = SIZE1024;
    size_t alignment = 1 << 12; /* 4096 bytes alignment */
    int *ptr = NULL;
    int *ptr1;

    /* Allocate memory with 4096 bytes alignment */

    if (posix_memalign((void **)&ptr, alignment, sizeof(int)) != 0)
    {
        /* Handle error */
      }

    /*Reallocate memory without using the original alignment.
    ptr1 may not be 4096 bytes aligned. */

    ptr1 = (int *)realloc(ptr, sizeof(int) * resize); //Noncompliant

    if (ptr1 == NULL)
    {
        /* Handle error */
    }

    /* Processing using ptr1 */

    /* Free before exit */
    free(ptr1);
}
```

In this example, the allocated memory is 4096-bytes aligned. `realloc()` then resizes the allocated memory. The new pointer `ptr1` might not be 4096-bytes aligned.

**Correction — Specify the Alignment for the Reallocated Memory**

When you reallocate the memory, use `posix_memalign()` and pass the alignment argument that you used to allocate the original memory.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE1024 1024

void func(void)
{
    size_t resize = SIZE1024;
    size_t alignment = 1 << 12; /* 4096 bytes alignment */
    int *ptr = NULL;

    /* Allocate memory with 4096 bytes alignment */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int)) != 0)
    {
        /* Handle error */
    }

    /* Reallocate memory using the original alignment. */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int) * resize) != 0)
    {
        /* Handle error */
        free(ptr);
        ptr = NULL;
    }

    /* Processing using ptr */

    /* Free before exit */
    free(ptr);
}
```

## Check Information

**Group:** Rule 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MEM36-C

# CERT C: Rule FIO30-C

Exclude user input from format strings

## Description

### Rule Definition

*Exclude user input from format strings.*

### Polyspace Implementation

The rule checker checks for **Tainted string format**.

### Extend Checker

A default Bug Finder analysis might not flag a **Tainted string format** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Tainted string format

#### Issue

**Tainted string format** detects string formatting with `printf`-style functions that contain elements from unsecure sources.

#### Risk

If you use externally controlled elements to format a string, you can cause buffer overflow or data-representation problems. An attacker can use these string formatting elements to view the contents of a stack using `%x` or write to a stack using `%n`.

#### Fix

Pass a static string to format string functions. This fix ensures that an external actor cannot control the string.

Another possible fix is to allow only the expected number of arguments. If possible, use functions that do not support the vulnerable `%n` operator in format strings.

#### Example - Get Elements from User Input

```
#include <stdio.h>
#include <unistd.h>
#define MAX 40
void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf(userstr);    //Noncompliant
}
```

This example prints the input argument `userstr`. The string is unknown. If it contains elements such as `%`, `printf` can interpret `userstr` as a string format instead of a string, causing your program to crash.

#### Correction — Print as String

One possible correction is to print `userstr` explicitly as a string so that there is no ambiguity.

```
#include "stdio.h"
#include <unistd.h>
#define MAX 40

void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf("%.20s", userstr);
}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO30-C

# CERT C: Rule FIO32-C

Do not perform operations on devices that are only appropriate for files

## Description

### Rule Definition

*Do not perform operations on devices that are only appropriate for files.*

### Polyspace Implementation

The rule checker checks for **Inappropriate I/O operation on device files**.

## Examples

### Inappropriate I/O operation on device files

#### Issue

**Inappropriate I/O operation on device files** occurs when you do not check whether a file name parameter refers to a device file before you pass it to these functions:

- `fopen()`
- `fopen_s()`
- `freopen()`
- `remove()`
- `rename()`
- `CreateFile()`
- `CreateFileA()`
- `CreateFileW()`
- `_wfopen()`
- `_wfopen_s()`

Device files are files in a file system that provide an interface to device drivers. You can use these files to interact with devices.

**Inappropriate I/O operation on device files** does not raise a defect when:

- You use `stat` or `lstat`-family functions to check the file name parameter before calling the previously listed functions.
- You use a string comparison function to compare the file name against a list of device file names.

#### Risk

Operations appropriate only for regular files but performed on device files can result in denial-of-service attacks, other security vulnerabilities, or system failures.

### Fix

Before you perform an I/O operation on a file:

- Use `stat()`, `lstat()`, or an equivalent function to check whether the file name parameter refers to a regular file.
- Use a string comparison function to compare the file name against a list of device file names.

#### Example - Using `fopen()` Without Checking `file_name`

```
#include <stdio.h>
#include <string.h>

#define SIZE1024 1024

FILE* func()
{

    FILE* f;
    const char file_name[SIZE1024] = "./tmp/file";

    if ((f = fopen(file_name, "w")) == NULL) { //Noncompliant
        /*handle error */
    };
    /*operate on file */
}
```

In this example, `func()` operates on the file `file_name` without checking whether it is a regular file. If `file_name` is a device file, attempts to access it can result in a system failure.

#### Correction — Check File with `lstat()` Before Calling `fopen()`

One possible correction is to use `lstat()` and the `S_ISREG` macro to check whether the file is a regular file. This solution contains a `TOCTOU` race condition that can allow an attacker to modify the file after you check it but before the call to fopen(). To prevent this vulnerability, ensure that `file_name` refers to a file in a secure folder.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>

#define SIZE1024 1024

FILE* func()
{

    FILE* f;
    const char file_name[SIZE1024] = "./tmp/file";
    struct stat orig_st;
    if ((lstat(file_name, &orig_st) != 0) ||
        (!S_ISREG(orig_st.st_mode))) {
        exit(0);
    }
    if ((f = fopen(file_name, "w")) == NULL) {
        /*handle error */
    };
    /*operate on file */
}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FIO32-C

# CERT C: Rule FIO34-C

Distinguish between characters read from a file and EOF or WEOF

## Description

### Rule Definition

*Distinguish between characters read from a file and EOF or WEOF.*

### Polyspace Implementation

The rule checker checks for **Character value absorbed into EOF**.

## Examples

### Character value absorbed into EOF

#### Issue

**Character value absorbed into EOF** occurs when you perform a data type conversion that makes a valid character value indistinguishable from `EOF` (End-of-File). Bug Finder flags the defect in one of the following situations:

- *End-of-File*: You perform a data type conversion such as from `int` to `char` that converts a non-EOF character value into `EOF`.

  ```
  char ch = (char)getchar()
  ```

  You then compare the result with EOF.

  ```
  if((int)ch == EOF)
  ```

  The conversion can be explicit or implicit.
- *Wide End-of-File*: You perform a data type conversion that can convert a non-WEOF wide character value into WEOF, and then compare the result with WEOF.

#### Risk

The data type `char` cannot hold the value `EOF` that indicates the end of a file. Functions such as `getchar` have return type `int` to accommodate `EOF`. If you convert from `int` to `char`, the values `UCHAR_MAX` (a valid character value) and `EOF` get converted to the same value -1 and become indistinguishable from each other. When you compare the result of this conversion with `EOF`, the comparison can lead to false detection of `EOF`. This rationale also applies to wide character values and `WEOF`.

#### Fix

Perform the comparison with `EOF` or `WEOF` before conversion.

#### Example - Return Value of `getchar` Converted to `char`

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define fatal_error() abort()

char func(void)
{
    char ch;
    ch = getchar();
    if (EOF == (int)ch) { //Noncompliant
        fatal_error();
    }
    return ch;
}
```

In this example, the return value of `getchar` is implicitly converted to `char`. If `getchar` returns `UCHAR_MAX`, it is converted to -1, which is indistinguishable from EOF. When you compare with EOF later, it can lead to a false positive.

#### Correction — Perform Comparison with EOF Before Conversion

One possible correction is to first perform the comparison with EOF, and then convert from `int` to `char`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

char func(void)
{
    int i;
    i = getchar();
    if (EOF == i) {
        fatal_error();
    }
    else {
        return (char)i;
    }
}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FIO34-C

# CERT C: Rule FIO37-C

Do not assume that fgets() or fgetws() returns a nonempty string when successful

## Description

### Rule Definition

*Do not assume that fgets() or fgetws() returns a nonempty string when successful.*

### Polyspace Implementation

The rule checker checks for **Use of indeterminate string**.

## Examples

### Use of indeterminate string

#### Issue

**Use of indeterminate string** occurs when you do not check the validity of the buffer returned from `fgets`-family functions. The checker raises a defect when such a buffer is used as:

- An argument in standard functions that print or manipulate strings or wide strings.
- A return value.
- An argument in external functions with parameter type `const char *` or `const wchar_t *`.

#### Risk

If an `fgets`-family function fails, the content of its output buffer is indeterminate. Use of such a buffer has undefined behavior and can result in a program that stops working or other security vulnerabilities.

#### Fix

Reset the output buffer of an `fgets`-family function to a known string value when the function fails.

#### Example - Output of `fgets()` Passed to External Function

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func(void) {
    char buf[SIZE20];

    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* 'buf' may contain an indeterminate string.  */
        ;
    }
    /* 'buf passed to external function */
    display_text(buf);  //Noncompliant
}
```

In this example, the output `buf` is passed to the external function `display_text()`, but its value is not reset if `fgets()` fails.

#### Correction — Reset `fgets()` Output on Failure

If `fgets()` fails, reset `buf` to a known value before you pass it to an external function.

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func1(void) {
    char buf[SIZE20];
    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* value of 'buf' reset after fgets() failure. */
        buf[0] = '\0';
    }
    /* 'buf' passed to external function */
    display_text(buf);
}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO37-C

# CERT C: Rule FIO38-C

Do not copy a FILE object

## Description

### Rule Definition

*Do not copy a FILE object.*

### Polyspace Implementation

The rule checker checks for **Misuse of a FILE object**.

## Examples

### Misuse of a FILE object

#### Issue

**Misuse of a FILE object** occurs when:

- You dereference a pointer to a FILE object, including indirect dereference by using `memcmp()`.
- You modify an entire FILE object or one of its components through its pointer.
- You take the address of FILE object that was not returned from a call to an `fopen`-family function. No defect is raised if a macro defines the pointer as the address of a built-in FILE object, such as `#define ptr (&__stdout)`.

#### Risk

In some implementations, the address of the pointer to a FILE object used to control a stream is significant. A pointer to a copy of a FILE object is interpreted differently than a pointer to the original object, and can potentially result in operations on the wrong stream. Therefore, the use of a copy of a FILE object can cause the software to stop responding, which an attacker might exploit in denial-of-service attacks.

#### Fix

Do not make a copy of a FILE object. Do not use the address of a FILE object that was not returned from a successful call to an `fopen`-family function.

#### Example - Copy of FILE Object Used in `fputs()`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

void fatal_error(void);

int func(void)
{
    /*'stdout' dereferenced and contents
        copied to 'my_stdout'. */
    FILE my_stdout = *stdout;   //Noncompliant

    /* Address of 'my_stdout' may not point to correct stream. */
    if (fputs("Hello, World!\n", &my_stdout) == EOF)   //Noncompliant
    {
        /* Handler error */
        fatal_error();
    }
    return 0;
}
```

In this example, FILE object `stdout` is dereferenced and its contents are copied to `my_stdout`. The contents of `stdout` might not be significant. `fputs()` is then called with the address of `my_stdout` as an argument. Because no call to `fopen()` or a similar function was made, the address of `my_stdout` might not point to the correct stream.

**Correction — Copy the FILE Object Pointer**

Declare `my_stdout` to point to the same address as `stdout` to ensure that you write to the correct stream when you call `fputs()`.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

void fatal_error(void);

int func(void)
{
    /* 'my_stdout' and 'stdout' point to the same object. */
    FILE *my_stdout = stdout;
    if (fputs("Hello, World!\n", my_stdout) == EOF)
    {
        /* Handler error */
        fatal_error();
    }
    return 0;
}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FIO38-C

# CERT C: Rule FIO39-C

Do not alternately input and output from a stream without an intervening flush or positioning call

## Description

### Rule Definition

*Do not alternately input and output from a stream without an intervening flush or positioning call.*

### Polyspace Implementation

The rule checker checks for **Alternating input and output from a stream without flush or positioning call**.

## Examples

### Alternating input and output from a stream without flush or positioning call

#### Issue

**Alternating input and output from a stream without flush or positioning call** occurs when:

- You do not perform a flush or function positioning call between an output operation and a following input operation on a file stream in update mode.
- You do not perform a function positioning call between an input operation and a following output operation on a file stream in update mode.

#### Risk

Alternating input and output operations on a stream without an intervening flush or positioning call is undefined behavior.

#### Fix

Call `fflush()` or a file positioning function such as `fseek()` or `fsetpos()` between output and input operations on an update stream.

Call a file positioning function between input and output operations on an update stream.

#### Example - Read After Write Without Intervening Flush

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";

void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;
    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Read operation after write without
    intervening flush. */
    if (fread(data, 1, SIZE20, file) < SIZE20)   //Noncompliant
      {
          (void)fclose(file);
          /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

In this example, the file `demo.txt` is opened for reading and appending. After the call to `fwrite()`, a call to `fread()` without an intervening flush operation is undefined behavior.

### Correction — Call `fflush()` Before the Read Operation

After writing data to the file, before calling `fread()`, perform a flush call.

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";


void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
```

```
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Buffer flush after write and before read */
    if (fflush(file) != 0)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO39-C

# CERT C: Rule FIO40-C

Reset strings on fgets() or fgetws() failure

## Description

### Rule Definition

*Reset strings on fgets() or fgetws() failure.*

### Polyspace Implementation

The rule checker checks for **Use of indeterminate string**.

## Examples

### Use of indeterminate string

#### Issue

**Use of indeterminate string** occurs when you do not check if a write operation using an `fgets`-family function such as:

```
char * fgets(char* buf, int n, FILE *stream);
```

succeeded and the buffer written has valid content, or you do not reset the buffer on failure. You then perform an operation that assumes a buffer with valid content. For instance, if the buffer with possibly indeterminate content is `buf` (as shown above), the checker raises a defect if:

- You pass `buf` as argument to standard functions that print or manipulate strings or wide strings.
- You return `buf` from a function.
- You pass `buf` as argument to external functions with parameter type `const char *` or `const wchar_t *`.
- You read `buf` as `buf[index]` or `*(buf + offset)`, where `index` or `offset` is a numerical value representing the distance from the beginning of the buffer.

#### Risk

If an `fgets`-family function fails, the content of its output buffer is indeterminate. Use of such a buffer has undefined behavior and can result in a program that stops working or other security vulnerabilities.

#### Fix

Reset the output buffer of an `fgets`-family function to a known string value when the function fails.

#### Example - Output of `fgets()` Passed to External Function

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>
```

```
#define SIZE20 20

extern void display_text(const char *txt);

void func(void) {
    char buf[SIZE20];

    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* 'buf' may contain an indeterminate string.  */
        ;
    }
    /* 'buf passed to external function */
    display_text(buf);  //Noncompliant
}
```

In this example, the output `buf` is passed to the external function `display_text()`, but its value is not reset if `fgets()` fails.

**Correction — Reset `fgets()` Output on Failure**

If `fgets()` fails, reset `buf` to a known value before you pass it to an external function.

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func1(void) {
    char buf[SIZE20];
    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* value of 'buf' reset after fgets() failure. */
        buf[0] = '\0';
    }
    /* 'buf' passed to external function */
    display_text(buf);
}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**
“Check for and Review Coding Standard Violations”

**External Websites**
FIO40-C

# CERT C: Rule FIO41-C

Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects

## Description

### Rule Definition

*Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects.*

### Polyspace Implementation

The rule checker checks for **Stream argument with possibly unintended side effects**.

## Examples

### Stream argument with possibly unintended side effects

#### Issue

**Stream argument with possibly unintended side effects** occurs when you call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects.

**Stream argument with possibly unintended side effects** considers the following as stream side effects:

- Any assignment of a variable of a stream, such as FILE *, or any assignment of a variable of a deeper stream type, such as an array of FILE *.
- Any call to a function that manipulates a stream or a deeper stream type.

The number of defects raised corresponds to the number of side effects detected. When a stream argument is evaluated multiple times in a function implemented as a macro, a defect is raised for each evaluation that has a side effect.

A defect is also raised on functions that are not implemented as macros but that can be implemented as macros on another operating system.

#### Risk

If the function is implemented as an unsafe macro, the stream argument can be evaluated more than once, and the stream side effect happens multiple times. For instance, a stream argument calling fopen() might open the same file multiple times, which is unspecified behavior.

#### Fix

To ensure that the side effect of a stream happens only once, use a separate statement for the stream argument.

#### Example - Stream Argument of getc() Has Side Effect fopen()

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#define fatal_error() abort()

const char* myfile = "my_file.log";

void func(void)
{
    int c;
    FILE* fptr;
    /* getc() has stream argument fptr with
    * 2 side effects: call to fopen(), and assignment
    * of fptr
    */
    c = getc(fptr = fopen(myfile, "r")); //Noncompliant
    if (c == EOF) {
        /* Handle error */
        (void)fclose(fptr);
        fatal_error();
    }
    if (fclose(fptr) == EOF) {
        /* Handle error */
        fatal_error();
    }
}

void main(void)
{
    func();

}
```

In this example, `getc()` is called with stream argument `fptr`. The stream argument has two side effects: the call to `fopen()` and the assignment of `fptr`. If `getc()` is implemented as an unsafe macro, the side effects happen multiple times.

#### Correction — Use Separate Statement for `fopen()`

One possible correction is to use a separate statement for `fopen()`. The call to `fopen()` and the assignment of `fptr` happen in this statement so there are no side effects when you pass `fptr` to `getc()`.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

#define fatal_error() abort()


const char* myfile = "my_file.log";

void func(void)
{
    int c;
    FILE* fptr;

    /* Separate statement for fopen()
    * before call to getc()
    */
    fptr = fopen(myfile, "r");
    if (fptr == NULL) {
        /* Handle error */
        fatal_error();
    }
    c = getc(fptr);
    if (c == EOF) {
        /* Handle error */
        (void)fclose(fptr);
        fatal_error();
    }
    if (fclose(fptr) == EOF) {
        /* Handle error */
        fatal_error();
    }
}

void main(void)
{
    func();

}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C (-cert-c)

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO41-C

# CERT C: Rule FIO42-C

Close files when they are no longer needed

## Description

### Rule Definition

*Close files when they are no longer needed.*

### Polyspace Implementation

The rule checker checks for **Resource leak**.

## Examples

### Resource leak

#### Issue

**Resource leak** occurs when you open a file stream by using a FILE pointer but do not close it before:

- The end of the pointer’s scope.
- Assigning the pointer to another stream.

#### Risk

If you do not release file handles explicitly as soon as possible, a failure can occur due to exhaustion of resources.

#### Fix

Close a FILE pointer before the end of its scope, or before you assign the pointer to another stream.

#### Example - FILE Pointer Not Released Before End of Scope

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" ); //Noncompliant
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

In this example, the file pointer fp1 is pointing to a file data1.txt. Before fp1 is explicitly dissociated from the file stream of data1.txt, it is used to access another file data2.txt.

#### Correction — Release FILE Pointer

One possible correction is to explicitly dissociate fp1 from the file stream of data1.txt.

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );
    fclose(fp1);

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FIO42-C

# CERT C: Rule FIO44-C

Only use values for fsetpos() that are returned from fgetpos()

## Description

### Rule Definition

*Only use values for fsetpos() that are returned from fgetpos().*

### Polyspace Implementation

The rule checker checks for **Invalid file position**.

## Examples

### Invalid file position

#### Issue

**Invalid file position** occurs when the file position argument of `fsetpos()` uses a value that is not obtained from `fgetpos()`.

#### Risk

The function `fgetpos(FILE *stream, fpos_t *pos)` gets the current file position of the stream. When you use any other value as the file position argument of `fsetpos(FILE *stream, const fpos_t *pos)`, you might access an unintended location in the stream.

#### Fix

Use the value returned from a successful call to `fgetpos()` as the file position argument of `fsetpos()`.

#### Example - `memset()` Sets File Position Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset' */
    (void)memset(&offset, 0, sizeof(offset));

    /* Read data from file */

    /* Return to the initial position. offset was not
    returned from a call to fgetpos()    */
    if (fsetpos(file, &offset) != 0)           //Noncompliant
    {
        /* Handle error */
    }
    return file;
}
```

In this example, `fsetpos()` uses `offset` as its file position argument. However, the value of `offset` is set by `memset()`. The preceding code might access the wrong location in the stream.

**Correction — Use a File Position Returned From `fgetpos()`**

Call `fgetpos()`, and if it returns successfully, use the position argument in your call to `fsetpos()`.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset'
    using fgetpos() */
    if (fgetpos(file, &offset) != 0)
    {
        /* Handle error */
    }

    /* Read data from file */

    /* Back to the initial position */
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

FIO44-C

# CERT C: Rule FIO45-C

Avoid TOCTOU race conditions while accessing files

## Description

### Rule Definition

*Avoid TOCTOU race conditions while accessing files.*

### Polyspace Implementation

The rule checker checks for **File access between time of check and use (TOCTOU)**.

## Examples

### File access between time of check and use (TOCTOU)

#### Issue

**File access between time of check and use (TOCTOU)** detects race condition issues between checking the existence of a file or folder, and using a file or folder.

#### Risk

An attacker can access and manipulate your file between your check for the file and your use of a file. Symbolic links are particularly risky because an attacker can change where your symbolic link points.

#### Fix

Before using a file, do not check its status. Instead, use the file and check the results afterward.

If you want to justify this violation, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Check File Before Using

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    if (access(log_path, W_OK)==0) {
        FILE* f = fopen(log_path, "w"); //Noncompliant
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

In this example, before opening and using the file, the function checks if the file exists. However, an attacker can change the file between the first and second lines of the function.

#### Correction — Open Then Check

One possible correction is to open the file, and then check the existence and contents afterward.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    int fd = open(log_path, O_WRONLY);
    if (fd!=-1) {
        FILE *f = fdopen(fd, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FIO45-C

# CERT C: Rule FIO46-C

Do not access a closed file

## Description

### Rule Definition

*Do not access a closed file.*

### Polyspace Implementation

The rule checker checks for **Use of previously closed resource**.

## Examples

### Use of previously closed resource

#### Issue

**Use of previously closed resource** occurs when a function operates on a stream that you closed earlier in your code.

#### Risk

The standard states that the value of a `FILE*` pointer is indeterminate after you close the stream associated with it. Operations using the `FILE*` pointer can produce unintended results.

#### Fix

One possible fix is to close the stream only at the end of operations. Another fix is to reopen the stream before using it again.

#### Example - Use of FILE* Pointer After Closing Stream

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fclose(fp);
        fprintf(fp,"text"); //Noncompliant
    }
}
```

In this example, `fclose` closes the stream associated with `fp`. When you use `fprintf` on `fp` after `fclose`, the **Use of previously closed resource** defect appears.

#### Correction — Close Stream After All Operations

One possible correction is to reverse the order of the `fprintf` and `fclose` operations.

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fprintf(fp,"text");
        fclose(fp);
    }
}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C (-cert-c)

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO46-C

# CERT C: Rule FIO47-C

Use valid format strings

## Description

### Rule Definition

*Use valid format strings.*

### Polyspace Implementation

The rule checker checks for **Format string specifiers and arguments mismatch**.

## Examples

### Format string specifiers and arguments mismatch

#### Issue

**Format string specifiers and arguments mismatch** occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

#### Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

#### Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
  const char *message = "License not available";
  int err_number = -4;
  printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {
    unsigned long fst = 1;

    printf("%d\n", fst); //Noncompliant
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

#### Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

#### Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

## Check Information

**Group:** Rule 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO47-C

# CERT C: Rule ENV30-C

Do not modify the object referenced by the return value of certain functions

## Description

### Rule Definition

*Do not modify the object referenced by the return value of certain functions.*

### Polyspace Implementation

The rule checker checks for **Modification of internal buffer returned from nonreentrant standard function**.

## Examples

### Modification of internal buffer returned from nonreentrant standard function

#### Issue

**Modification of internal buffer returned from nonreentrant standard function** occurs when the following happens:

- A nonreentrant standard function returns a pointer.
- You attempt to write to the memory location that the pointer points to.

Nonreentrant standard functions that return a non `const`-qualified pointer to an internal buffer include `getenv`, `getlogin`, `crypt`, `setlocale`, `localeconv`, `strerror` and others.

#### Risk

Modifying the internal buffer that a nonreentrant standard function returns can cause the following issues:

- It is possible that the modification does not succeed or alters other internal data.

  For instance, `getenv` returns a pointer to an environment variable value. If you modify this value, you alter the environment of the process and corrupt other internal data.
- Even if the modification succeeds, it is possible that a subsequent call to the same standard function does not return your modified value.

  For instance, you modify the environment variable value that `getenv` returns. If another process, thread, or signal handler calls `setenv`, the modified value is overwritten. Therefore, a subsequent call to `getenv` does not return your modified value.

#### Fix

Avoid modifying the internal buffer using the pointer returned from the function.

#### Example - Modification of getenv Return Value

```
#include <stdlib.h>
#include <string.h>
```

```
void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        strncpy(env, "C", 1); //Noncompliant
        printstr(env);
    }
}
```

In this example, the first argument of `strncpy` is the return value from a nonreentrant standard function `getenv`. The behavior can be undefined because `strncpy` modifies this argument.

#### Correction - Copy Return Value of `getenv` and Modify Copy

One possible solution is to copy the return value of `getenv` and pass the copy to the `strncpy` function.

```
#include <stdlib.h>
#include <string.h>
enum {
    SIZE20 = 20
};

void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        char env_cp[SIZE20];
        strncpy(env_cp, env, SIZE20);
        strncpy(env_cp, "C", 1);
        printstr(env_cp);
    }
}
```

## Check Information

**Group:** Rule 10. Environment (ENV)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

ENV30-C

# CERT C: Rule ENV31-C

Do not rely on an environment pointer following an operation that may invalidate it

## Description

### Rule Definition

*Do not rely on an environment pointer following an operation that may invalidate it.*

### Polyspace Implementation

The rule checker checks for **Environment pointer invalidated by previous operation**.

## Examples

### Environment pointer invalidated by previous operation

#### Issue

**Environment pointer invalidated by previous operation** occurs when you use the third argument of *main()* in a hosted environment to access the environment after an operation modifies the environment. In a hosted environment, many C implementations support the nonstandard syntax:

```
main (int argc, char *argv[], char *envp[])
```

A call to a `setenv` or `putenv` family function modifies the environment pointed to by `*envp`.

#### Risk

When you modify the environment through a call to a `setenv` or `putenv` family function, the environment memory can potentially be reallocated. The hosted environment pointer is not updated and might point to an incorrect location. A call to this pointer can return unexpected results or cause an abnormal program termination.

#### Fix

Do not use the hosted environment pointer. Instead, use global external variable `environ` in Linux, `_environ` or `_wenviron` in Windows, or their equivalent. When you modify the environment, these variables are updated.

#### Example - Access Environment Through Pointer envp

```
#include <stdio.h>
#include <stdlib.h>

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

/* envp is from main function */
int func(char **envp)
{
    /* Call to setenv may cause environment
     *memory to be reallocated
     */
    if (setenv(("MY_NEW_VAR"),("new_value"),1) != 0)
    {
        /* Handle error */
        return -1;
    }
    /* envp not updated after call to setenv, and may
     *point to incorrect location.
     **/
    if (envp != ((void *)0)) {  //Noncompliant
        use_envp(envp);
/* No defect on second access to
*envp because defect already raised */
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func(envp);
    }
}
```

In this example, `envp` is accessed inside `func()` after a call to `setenv` that can reallocate the environment memory. `envp` can point to an incorrect location because it is not updated after `setenv` modifies the environment. No defect is raised when `use_envp()` is called because the defect is already raised on the previous line of code.

#### Correction — Use Global External Variable `environ`

One possible correction is to access the environment by using a variable that is always updated after a call to `setenv`. For instance, in the following code, the pointer `envp` is still available from `main()`, but the environment is accessed in `func()` through the global external variable `environ`.

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

int func(void)
{
    if (setenv(("MY_NEW_VAR"), ("new_value"),1) != 0) {
        /* Handle error */
        return -1;
    }
  /* Use global external variable environ
   *which is always updated after a call to setenv */

    if (environ != NULL) {
        use_envp(environ);
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
```

```
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func();
    }
}
```

## Check Information

**Group:** Rule 10. Environment (ENV)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

ENV31-C

# CERT C: Rule ENV32-C

All exit handlers must return normally

## Description

### Rule Definition

*All exit handlers must return normally.*

### Polyspace Implementation

The rule checker checks for **Abnormal termination of exit handler**.

## Examples

### Abnormal termination of exit handler

#### Issue

**Abnormal termination of exit handler** looks for registered exit handlers. Exit handlers are registered with specific functions such as `atexit`, (WinAPI) `_onexit`, or `at_quick_exit()`. If the exit handler calls a function that interrupts the program's expected termination sequence, Polyspace raises a defect. Some functions that can cause abnormal exits are `exit`, `abort`, `longjmp`, or (WinAPI) `_onexit`.

#### Risk

If your exit handler terminates your program, you can have undefined behavior. Abnormal program termination means other exit handlers are not invoked. These additional exit handlers may do additional clean up or other required termination steps.

#### Fix

In inside exit handlers, remove calls to functions that prevent the exit handler from terminating normally.

#### Example - Exit Handler With Call to exit

```
#include <stdlib.h>

volatile int some_condition = 1;
void demo_exit1(void)
{
    /* ... Cleanup code ... */
    return;
}
void exitabnormalhandler(void)
{
    if (some_condition)
    {
        /* Clean up */
        exit(0); //Noncompliant
    }
    return;
}

int demo_install_exitabnormalhandler(void)
{

    if (atexit(demo_exit1) != 0) /* demo_exit1() performs additional cleanup */
    {
        /* Handle error */
    }
    if (atexit(exitabnormalhandler) != 0)
    {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

In this example, demo_install_exitabnormalhandler registers two exit handlers, demo_exit1 and exitabnormalhandler. Exit handlers are invoked in the reverse order of which they are registered. When the program ends, exitabnormalhandler runs, then demo_exit1. However, exitabnormalhandler calls exit interrupting the program exit process. Having this exit inside an exit handler causes undefined behavior because the program is not finished cleaning up safely.

#### Correction — Remove exit from Exit Handler

One possible correction is to let your exit handlers terminate normally. For this example, exit is removed from exitabnormalhandler, allowing the exit termination process to complete as expected.

```
#include <stdlib.h>

volatile int some_condition = 1;
void demo_exit1(void)
{
    /* ... Cleanup code ... */
    return;
}
void exitabnormalhandler(void)
{
    if (some_condition)
    {
        /* Clean up */
        /* Return normally */
    }
    return;
}

int demo_install_exitabnormalhandler(void)
{

    if (atexit(demo_exit1) != 0) /* demo_exit1() continues clean up */
    {
        /* Handle error */
    }
    if (atexit(exitabnormalhandler) != 0)
    {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

## Check Information

**Group:** Rule 10. Environment (ENV)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ENV32-C

# CERT C: Rule ENV33-C

Do not call system()

## Description

### Rule Definition

*Do not call system().*

### Polyspace Implementation

The rule checker checks for **Unsafe call to a system function**.

## Examples

### Unsafe call to a system function

#### Issue

**Unsafe call to a system function** occurs when you use a function that invokes an implementation-defined command processor. These functions include:

- The C standard `system()` function.
- The POSIX `popen()` function.
- The Windows `_popen()` and `_wpopen()` functions.

#### Risk

If the argument of a function that invokes a command processor is not sanitized, it can cause exploitable vulnerabilities. An attacker can execute arbitrary commands or read and modify data anywhere on the system.

#### Fix

Do not use a `system`-family function to invoke a command processor. Instead, use safer functions such as POSIX `execve()` and WinAPI `CreateProcess()`.

#### Example - `system()` Called

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};

void func(char *arg)
{
    char buf[SIZE512];
    int retval=sprintf(buf, "/usr/bin/any_cmd %s", arg);
    if (retval<=0 || retval>SIZE512){
        /* Handle error */
        abort();
    }
    /* Use of system() to pass any_cmd with
    unsanitized argument to command processor */

    if (system(buf) == -1) { //Noncompliant
    /* Handle error */
  }
}
```

In this example, `system()` passes its argument to the host environment for the command processor to execute. This code is vulnerable to an attack by command-injection.

**Correction — Sanitize Argument and Use execve()**

In the following code, the argument of `any_cmd` is sanitized, and then passed to `execve()` for execution. `exec`-family functions are not vulnerable to command-injection attacks.

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};


void func(char *arg)
{
  char *const args[SIZE3] = {"any_cmd", arg, NULL};
  char  *const env[] = {NULL};

  /* Sanitize argument */

  /* Use execve() to execute any_cmd. */

  if (execve("/usr/bin/time", args, env) == -1) {
    /* Handle error */
  }
}
```

## Check Information

**Group:** Rule 10. Environment (ENV)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

**External Websites**
ENV33-C

# CERT C: Rule ENV34-C

Do not store pointers returned by certain functions

## Description

### Rule Definition

*Do not store pointers returned by certain functions.*

### Polyspace Implementation

The rule checker checks for **Misuse of return value from nonreentrant standard function**.

## Examples

### Misuse of return value from nonreentrant standard function

#### Issue

**Misuse of return value from nonreentrant standard function** occurs when these events happen in this sequence:

1. You point to the buffer returned from a nonreentrant standard function such as `getenv` or `setlocale`.

   ```
   user = getenv("USER");
   ```
2. You call that nonreentrant standard function again.

   ```
   user2 = getenv("USER2");
   ```
3. You use or dereference the pointer from the first step expecting the buffer to remain unmodified since that step. In the meantime, the call in the second step has modified the buffer.

   For instance:

   ```
   var=*user;
   ```

In some cases, the defect might appear even if you do not call the `getenv` function a second time but simply return the pointer. For instance:

```
char* func() {
     user=getenv("USER");
     .
     .
     return user;
}
```

For information on which functions are covered by this defect, see documentation on nonreentrant standard functions.

#### Risk

The C Standard allows nonreentrant functions such as `getenv` to return a pointer to a *static* buffer. Because the buffer is static, a second call to `getenv` modifies the buffer. If you continue to use the

pointer returned from the first call past the second call, you can see unexpected results. The buffer that it points to no longer has values from the first call.

The defect appears even if you do not call `getenv` a second time but simply return the pointer. The reason is that someone calling your function might use the returned pointer *after* a second call to `getenv`. By returning the pointer from your call to `getenv`, you make your function unsafe to use.

The same rationale is true for other nonreentrant functions covered by this defect.

**Fix**

After the first call to `getenv`, make a copy of the buffer that the returned pointer points to. After the second call to `getenv`, use this copy. Even if the second call modifies the buffer, your copy is untouched.

**Example - Return from `getenv` Used After Second Call to `getenv`**

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");   /* First call */
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');

        if (user_name_from_home != NULL) {
            user = getenv("USER");   /* Second call */
            if ((user != NULL) &&
                (strcmp(user, user_name_from_home) == 0))  //Noncompliant
            {
                result = 1;
            }
        }
    }
    return result;
}
```

In this example, the pointer `user_name_from_home` is derived from the pointer `home`. `home` points to the buffer returned from the first call to `getenv`. Therefore, `user_name_from_home` points to a location in the same buffer.

After the second call to `getenv`, the buffer is modified. If you continue to use `user_name_from_home`, you can get unexpected results.

**Correction — Make Copy of Buffer Before Second Call**

If you want to access the buffer from the first call to `getenv` past the second call, make a copy of the buffer after the first call. One possible correction is to use the `strdup` function to make the copy.

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');
        if (user_name_from_home != NULL) {
            /* Make copy before second call */
            char *saved_user_name_from_home = strdup(user_name_from_home);
            if (saved_user_name_from_home != NULL) {
                user = getenv("USER");
                if ((user != NULL) &&
                    (strcmp(user, saved_user_name_from_home) == 0))
                {
                    result = 1;
                }
                free(saved_user_name_from_home);
            }
        }
    }
    return result;
}
```

## Check Information

**Group:** Rule 10. Environment (ENV)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

ENV34-C

# CERT C: Rule SIG30-C

Call only asynchronous-safe functions within signal handlers

## Description

### Rule Definition

*Call only asynchronous-safe functions within signal handlers.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Function called from signal handler not asynchronous-safe**.
- **Function called from signal handler not asynchronous-safe (strict)**.

## Examples

### Function called from signal handler not asynchronous-safe

#### Issue

**Function called from signal handler not asynchronous-safe** occurs when a signal handler calls a function that is not asynchronous-safe according to the POSIX standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

#### Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

#### Fix

The POSIX standard defines these functions as asynchronous-safe. You can call these functions from a signal handler.

| | | |
|---|---|---|
| `_exit()` | `getpgrp()` | `setsockopt()` |
| `_Exit()` | `getpid()` | `setuid()` |
| `abort()` | `getppid()` | `shutdown()` |
| `accept()` | `getsockname()` | `sigaction()` |
| `access()` | `getsockopt()` | `sigaddset()` |
| `aio_error()` | `getuid()` | `sigdelset()` |

| | | |
|---|---|---|
| aio_return() | kill() | sigemptyset() |
| aio_suspend() | link() | sigfillset() |
| alarm() | linkat() | sigismember() |
| bind() | listen() | signal() |
| cfgetispeed() | lseek() | sigpause() |
| cfgetospeed() | lstat() | sigpending() |
| cfsetispeed() | mkdir() | sigprocmask() |
| cfsetospeed() | mkdirat() | sigqueue() |
| chdir() | mkfifo() | sigset() |
| chmod() | mkfifoat() | sigsuspend() |
| chown() | mknod() | sleep() |
| clock_gettime() | mknodat() | sockatmark() |
| close() | open() | socket() |
| connect() | openat() | socketpair() |
| creat() | pathconf() | stat() |
| dup() | pause() | symlink() |
| dup2() | pipe() | symlinkat() |
| execl() | poll() | sysconf() |
| execle() | posix_trace_event() | tcdrain() |
| execv() | pselect() | tcflow() |
| execve() | pthread_kill() | tcflush() |
| faccessat() | pthread_self() | tcgetattr() |
| fchdir() | pthread_sigmask() | tcgetpgrp() |
| fchmod() | quick_exit() | tcsendbreak() |
| fchmodat() | raise() | tcsetattr() |
| fchown() | read() | tcsetpgrp() |
| fchownat() | readlink() | time() |
| fcntl() | readlinkat() | timer_getoverrun() |
| fdatasync() | recv() | timer_gettime() |
| fexecve() | recvfrom() | timer_settime() |
| fork() | recvmsg() | times() |
| fpathconf() | rename() | umask() |
| fstat() | renameat() | uname() |
| fstatat() | rmdir() | unlink() |
| fsync() | select() | unlinkat() |
| ftruncate() | sem_post() | utime() |
| futimens() | send() | utimensat() |

| | | |
|---|---|---|
| getegid() | sendmsg() | utimes() |
| geteuid() | sendto() | wait() |
| getgid() | setgid() | waitpid() |
| getgroups() | setpgid() | write() |
| getpeername() | setsid() | |

Functions not in the previous table are not asynchronous-safe, and should not be called from a signal hander.

### Example - Call to `printf()` Inside Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler(int signum)
{
    /* Call function printf() that is not
    asynchronous-safe */
    printf("signal %d received.", signum);  //Noncompliant
    e_flag = 1;
}

int main(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, sizeof(char));
    if (info == NULL)
    {
        /* Handle Error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

In this example, `sig_handler` calls `printf()` when catching a signal. If the handler catches another signal while `printf()` is executing, the behavior of the program is undefined.

**Correction — Set Flag Only in Signal Handler**

Use your signal handler to set only the value of a flag. `e_flag` is of type volatile `sig_atomic_t`. `sig_handler` can safely access it asynchronously.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler1(int signum)
{
    int s0 = signum;
    e_flag = 1;
}

int func(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler1) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, 1);
    if (info == NULL)
    {
        /* Handle error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

### Function called from signal handler not asynchronous-safe (strict)

#### Issue

**Function called from signal handler not asynchronous-safe (strict)** occurs when a signal handler calls a function that is not asynchronous-safe according to the C standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

When you select the checker **Function called from signal handler not asynchronous-safe**, the checker detects calls to functions that are not asynchronous-safe according to the POSIX standard. **Function called from signal handler not asynchronous-safe (strict)** does not raise a defect for these cases. **Function called from signal handler not asynchronous-safe (strict)** raises a defect for functions that are asynchronous-safe according to the POSIX standard but not according to the C standard.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

#### Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

#### Fix

The C standard defines the following functions as asynchronous-safe. You can call these functions from a signal handler:

- `abort()`
- `_Exit()`
- `quick_exit()`
- `signal()`

#### Example - Call to `raise()` Inside Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}

void sig_handler(int signum)
{
    int s0 = signum;
    /* Call raise() */
    if (raise(SIGTERM) != 0) {  //Noncompliant
        /* Handle error */
    }
}

int finc(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` calls `raise()` when catching a signal. If the handler catches another signal while `raise()` is executing, the behavior of the program is undefined.

#### Correction — Remove Call to `raise()` in Signal Handler

According to the C standard, the only functions that you can safely call from a signal handler are `abort()`, `_Exit()`, `quick_exit()`, and `signal()`.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}
void sig_handler(int signum)
{
    int s0 = signum;
}

int func(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

## Check Information

**Group:** Rule 11. Signals (SIG)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

SIG30-C

# CERT C: Rule SIG31-C

Do not access shared objects in signal handlers

## Description

### Rule Definition

*Do not access shared objects in signal handlers.*

### Polyspace Implementation

The rule checker checks for **Shared data access within signal handler**.

## Examples

### Shared data access within signal handler

#### Issue

**Shared data access within signal handler** occurs when you access or modify a shared object inside a signal handler.

#### Risk

When you define a signal handler function to access or modify a shared object, the handler accesses or modifies the shared object when it receives a signal. If another function is already accessing the shared object, that function causes a race condition and can leave the data in an inconsistent state.

#### Fix

To access or modify shared objects inside a signal handler, check that the objects are lock-free atomic, or, if they are integers, declare them as `volatile sig_atomic_t`.

#### Example - `int` Variable Access in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* declare global variable. */
int e_flag;

void sig_handler(int signum)
{
    /* Signal handler accesses variable that is not
    of type volatile sig_atomic_t. */
    e_flag = signum;  //Noncompliant
}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` accesses `e_flag`, a variable of type `int`. A concurrent access by another function can leave `e_flag` in an inconsistent state.

### Correction — Declare Variable of Type `volatile sig_atomic_t`

Before you access a shared variable from a signal handler, declare the variable with type `volatile sig_atomic_t` instead of `int`. You can safely access variables of this type asynchronously.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* Declare variable of type volatile sig_atomic_t. */
volatile sig_atomic_t e_flag;
void sig_handler(int signum)
{
    /* Use variable of proper type inside signal handler. */
    e_flag = signum;

}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

## Check Information

**Group:** Rule 11. Signals (SIG)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

SIG31-C

# CERT C: Rule SIG34-C

Do not call signal() from within interruptible signal handlers

## Description

### Rule Definition

*Do not call signal() from within interruptible signal handlers.*

### Polyspace Implementation

The rule checker checks for **Signal call from within signal handler**.

## Examples

### Signal call from within signal handler

#### Issue

**Signal call from within signal handler** occurs when you call the function `signal()` from a signal handler on Windows platforms.

The issue is detected only if you specify a Visual Studio compiler. See `Compiler (-compiler)`.

#### Risk

The function `signal()` associates a signal with a signal handler function. On platforms such as Windows, which removes this association after receiving the signal, you might call the function `signal()` again *within the signal handler* to re-establish the association.

However, this attempt to make a signal handler persistent is prone to race conditions. On Windows platforms, from the time the signal handler begins execution to when the `signal` function is called again, it is the default signal handling, `SIG_DFL`, that is active. If a second signal is received within this time window, you see the default signal handling and not the custom signal handler, but you might expect otherwise.

#### Fix

Do not call `signal()` from a signal handler on Windows platforms.

#### Example - `signal()` Called from Signal Handler

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;

void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;

    /* Call signal() to reestablish sig_handler
    upon receiving SIG_ERR. */

    if (signal(s0, sig_handler) == SIG_ERR)  //Noncompliant
    {
        /* Handle error */
    }
}

void func(void)
{
        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
  /* more code */
}
```

In this example, the definition of `sig_handler()` includes a call to `signal()` when the handler catches `SIG_ERR`. On Windows platforms, signal handlers are nonpersistent. This code can result in a race condition.

The issue is detected only if you specify a compiler such as `visual15.x` for the analysis.

**Correction — Do Not Call `signal()` from Signal Handler**

Avoid attempting to make a signal handler persistent on Windows. If your code requires the use of a persistent signal handler on a Windows platform, use a persistent signal handler after performing a thorough risk analysis.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;


void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;
    /* No call to signal() */
}

int main(void)
{

        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }
}
```

## Check Information

**Group:** Rule 11. Signals (SIG)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

SIG34-C

# CERT C: Rule SIG35-C

Do not return from a computational exception signal handler

## Description

### Rule Definition

*Do not return from a computational exception signal handler.*

### Polyspace Implementation

The rule checker checks for **Return from computational exception signal handler**.

## Examples

### Return from computational exception signal handler

#### Issue

**Return from computational exception signal handler** occurs when a signal handler returns after catching a computational exception signal SIGFPE, SIGILL, or SIGSEGV.

#### Risk

A signal handler that returns normally from a computational exception is undefined behavior. Even if the handler attempts to fix the error that triggered the signal, the program can behave unexpectedly.

#### Fix

Check the validity of the values of your variables before the computation to avoid using a signal handler to catch exceptions. If you cannot avoid a handler to catch computation exception signals, call abort(), quick_exit(), or _Exit() in the handler to stop the program.

#### Example - Signal Handler Return from Division by Zero

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

static volatile sig_atomic_t denom;
/* Declare signal handler to catch division by zero
computation error. */
void sig_handler(int s)
{
    int s0 = s;
    if (denom == 0)
    {
        denom = 1;
    }
    /* Normal return from computation exception
    signal */
    return;  //Noncompliant
}
```

```
long func(int v)
{
    denom = (sig_atomic_t)v;

        if (signal(SIGFPE, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }

    long result = 100 / (long)denom;
    return result;
}
```

In this example, `sig_handler` is declared to handle a division by zero computation error. The handler changes the value of `denom` if it is zero and returns, which is undefined behavior.

#### Correction — Call `abort()` to Terminate Program

After catching a computational exception, call `abort()` from `sig_handler` to exit the program without further error.

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

static volatile sig_atomic_t denom;
/* Declare signal handler to catch division by zero
computation error. */

void sig_handler(int s)
{
    int s0 = s;
    /* call to abort() to exit the program */
    abort();
}

long func(int v)
{
    denom = (sig_atomic_t)v;

        if (signal(SIGFPE, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }

    long result = 100 / (long)denom;
    return result;
}
```

## Check Information

**Group:** Rule 11. Signals (SIG)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

SIG35-C

# CERT C: Rule ERR30-C

Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure

## Description

### Rule Definition

*Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Misuse of errno**.
- **Errno not reset**.

## Examples

### Misuse of errno

#### Issue

**Misuse of errno** occurs when you check `errno` for error conditions in situations where checking `errno` does not guarantee the absence of errors. In some cases, checking `errno` can lead to false positives.

For instance, you check `errno` following calls to the functions:

- `fopen`: If you follow the ISO Standard, the function might not set `errno` on errors.
- `atof`: If you follow the ISO Standard, the function does not set `errno`.
- `signal`: The `errno` value indicates an error only if the function returns the `SIG_ERR` error indicator.

#### Risk

The ISO C Standard does not enforce that these functions set `errno` on errors. Whether the functions set `errno` or not is implementation-dependent.

To detect errors, if you check `errno` alone, the validity of this check also becomes implementation-dependent.

In some cases, the `errno` value indicates an error only if the function returns a specific error indicator. If you check `errno` before checking the function return value, you can see false positives.

#### Fix

For information on how to detect errors, see the documentation for that specific function.

Typically, the functions return an out-of-band error indicator to indicate errors. For instance:

- fopen returns a null pointer if an error occurs.
- signal returns the SIG_ERR error indicator and sets errno to a positive value. Check errno only after you have checked the function return value.

### Example - Incorrectly Checking for errno After fopen Call

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    errno = 0;
    fileptr = fopen(temp_filename, "w+b");
    if (errno != 0) { //Noncompliant
        if (fileptr != NULL) {
            (void)fclose(fileptr);
        }
        /* Handle error */
        fatal_error();
    }
    return fileptr;
}
```

In this example, errno is the first variable that is checked after a call to fopen. You might expect that fopen changes errno to a nonzero value if an error occurs. If you run this code with an implementation of fopen that does not set errno on errors, you might miss an error condition. In this situation, fopen can return a null pointer that escapes detection.

### Correction — Check Return Value of fopen After Call

One possible correction is to only check the return value of fopen for a null pointer.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    fileptr = fopen(temp_filename, "w+b");
    if (fileptr == NULL) {
        fatal_error();
    }
    return fileptr;
}
```

## Errno not reset

### Issue

**Errno not reset** occurs when you do not reset errno before calling a function that sets errno to indicate error conditions. However, you check errno for those error conditions after the function call.

### Risk

The errno is not clean and can contain values from a previous call. Checking errno for errors can give the false impression that an error occurred.

errno is set to zero at program startup but subsequently, errno is not reset by a C standard library function. You must explicitly set errno to zero when required.

### Fix

Before calling a function that sets errno to indicate error conditions, reset errno to zero explicitly.

### Example - errno Not Reset Before Call to strtod

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    f1 = strtod (s1, NULL);
    if (0 == errno) {         //Noncompliant
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
                  {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

In this example, errno is not reset to 0 before the first call to strtod. Checking errno for 0 later can lead to a false positive.

### Correction — Reset errno Before Call

One possible correction is to reset errno to 0 before calling strtod.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>
```

```
#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    errno = 0;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
              {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

## Check Information

**Group:** Rule 12. Error Handling (ERR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

ERR30-C

# CERT C: Rule ERR32-C

Do not rely on indeterminate values of errno

## Description

### Rule Definition

*Do not rely on indeterminate values of errno.*

### Polyspace Implementation

The rule checker checks for **Misuse of errno in a signal handler**.

## Examples

### Misuse of errno in a signal handler

#### Issue

**Misuse of errno in a signal handler** occurs when you call one of these functions in a signal handler:

- `signal`: You call the `signal` function in a signal handler and then read the value of `errno`.

  For instance, the signal handler function `handler` calls `signal` and then calls `perror`, which reads `errno`.

  ```
  typedef void (*pfv)(int);

  void handler(int signum) {
    pfv old_handler = signal(signum, SIG_DFL);
    if (old_handler == SIG_ERR) {
      perror("SIGINT handler");
    }
  }
  ```
- `errno`-setting POSIX function: You call an `errno`-setting POSIX function in a signal handler but do not restore `errno` when returning from the signal handler.

  For instance, the signal handler function `handler` calls `waitpid`, which changes `errno`, but does not restore `errno` before returning.

  ```
  #include <stddef.h>
  #include <errno.h>
  #include <sys/wait.h>

  void handler(int signum) {
    int rc = waitpid(-1, NULL, WNOHANG);
    if (ECHILD != errno) {
    }
  }
  ```

### Risk

In each case that the checker flags, you risk relying on an indeterminate value of errno.

- signal: If the call to signal in a signal handler fails, the value of errno is indeterminate (see C11 Standard, Sec. 7.14.1.1). If you rely on a specific value of errno, you can see unexpected results.
- errno-setting POSIX function: An errno-setting function sets errno on failure. If you read errno after a signal handler is called and the signal handler itself calls an errno-setting function, you can see unexpected results.

### Fix

Avoid situations where you risk relying on an indeterminate value of errno.

- signal: After calling the signal function in a signal handler, do not read errno or use a function that reads errno.
- errno-setting POSIX function: Before calling an errno-setting function in a signal handler, save errno to a temporary variable. Restore errno from this variable before returning from the signal handler.

### Example - Reading errno After signal Call in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#define fatal_error() abort()

void handler(int signum) {
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        perror("SIGINT handler");
    }
} //Noncompliant

int func(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
        fatal_error();
    }
    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
        fatal_error();
    }
    return 0;
}
```

In this example, the function handler is called to handle the SIGINT signal. In the body of handler, the signal function is called. Following this call, the value of errno is indeterminate. The checker raises a defect when the perror function is called because perror relies on the value of errno.

### Correction — Avoid Reading errno After signal Call

One possible correction is to not read errno after calling the signal function in a signal handler. The corrected code here calls the abort function via the fatal_error macro instead of the perror function.

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#define fatal_error() abort()

void handler(int signum) {
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        fatal_error();
    }
}

int func(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
        fatal_error();
    }
    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
        fatal_error();
    }
    return 0;
}
```

## Check Information

**Group:** Rule 12. Error Handling (ERR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

ERR32-C

# CERT C: Rule ERR33-C

Detect and handle standard library errors

## Description

### Rule Definition

*Detect and handle standard library errors.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Errno not checked**.
- **Returned value of a sensitive standard function not checked**.
- **Unprotected dynamic memory allocation**.
- **Pointer overwritten during reallocation**.

## Examples

### Errno not checked

#### Issue

**Errno not checked** occurs when you call a function that sets `errno` to indicate error conditions, but do not check `errno` after the call. For these functions, checking `errno` is the only reliable way to determine if an error occurred.

Functions that set `errno` on errors include:

- `fgetwc`, `strtol`, and `wcstol`.

  For a comprehensive list of functions, see documentation about errno.
- POSIX `errno`-setting functions such as `encrypt` and `setkey`.

#### Risk

To see if the function call completed without errors, check `errno` for error values.

The return values of these `errno`-setting functions do not indicate errors. The return value can be one of the following:

- `void`
- Even if an error occurs, the return value can be the same as the value from a successful call. Such return values are called in-band error indicators.

You can determine if an error occurred only by checking `errno`.

For instance, `strtol` converts a string to a long integer and returns the integer. If the result of conversion overflows, the function returns `LONG_MAX` and sets `errno` to `ERANGE`. However, the function can also return LONG_MAX from a successful conversion. Only by checking errno can you distinguish between an error and a successful conversion.

#### Fix

Before calling the function, set errno to zero.

After the function call, to see if an error occurred, compare errno to zero. Alternatively, compare errno to known error indicator values. For instance, strtol sets errno to ERANGE to indicate errors.

The error message in the Polyspace result shows the error indicator value that you can compare to.

#### Example - errno Not Checked After Call to strtol

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    long val = strtol(str, &endptr, base); //Noncompliant
    printf("Return value of strtol() = %ld\n", val);
}
```

You are using the return value of strtol without checking errno.

#### Correction — Check errno After Call

Before calling strtol, set errno to zero . After a call to strtol, check the return value for LONG_MIN or LONG_MAX and errno for ERANGE.

```
#include<stdlib.h>
#include<stdio.h>
#include<errno.h>
#include<limits.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    errno = 0;
    long val = strtol(str, &endptr, base);
    if((val == LONG_MIN || val == LONG_MAX) && errno == ERANGE) {
         printf("strtol error");
         exit(EXIT_FAILURE);
    }
    printf("Return value of strtol() = %ld\n", val);
}
```

### Returned value of a sensitive standard function not checked

#### Issue

**Returned value of a sensitive standard function not checked** occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `thrd_create`)
- Lock or unlock memory segments (for example, `mlock`)

#### Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

#### Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

#### Example - Sensitive Function Return Ignored

```
#include<stdio.h>
#include <wchar.h>
#include <locale.h>
void initialize(size_t n, size_t* size, wchar_t *wcs, const char *utf8) {

    scanf("%d",&n); //Noncompliant
    setlocale (LC_CTYPE, "en_US.UTF-8");   //Noncompliant
    *size = mbstowcs (wcs, utf8, n);
}
```

This example shows a call to the sensitive function `scanf()`. The return value of `scanf()` is ignored, causing a defect. Similarly, the pointer returned by `setlocale` is not checked. When `setlocal` returns a NULL pointer, the call to mbstowcs might fail or produce unexpected results. Polyspace flags these calls to sensitive functions when their returns are not checked.

#### Correction — Cast Function to (void)

One possible correction is to cast the functions to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of these sensitive functions.

```
#include<stdio.h>
#include <wchar.h>
#include <locale.h>
void initialize(size_t n, size_t* size, wchar_t *wcs, const char *utf8) {

    (void)scanf("%d",&n); //Compliant
    (void)setlocale (LC_CTYPE, "en_US.UTF-8");   //Compliant
    *size = mbstowcs (wcs, utf8, n);
}
```

#### Correction — Test Return Value

One possible correction is to test the return value of scanf and setlocale to check for errors.

```
#include<stdio.h>
#include <wchar.h>
#include <locale.h>
void initialize(size_t n, size_t* size, wchar_t *wcs, const char *utf8) {

    int flag = scanf("%d",&n);
    if(flag>0){ //Compliant
        // action
    }
    char* status = setlocale (LC_CTYPE, "en_US.UTF-8");
    if(status!=NULL){//Compliant
        *size = mbstowcs (wcs, utf8, n);
    }

}
```

#### Example - Critical Function Return Ignored

```
#include <threads.h>
int thrd_func(void);
void initialize() {
    thrd_t thr;
    int n = 1;

    (void) thrd_create(&thr,thrd_func,&n);  //Noncompliant
}
```

In this example, a critical function thrd_create is called and its return value is ignored by casting to void, but because thrd_create is a critical function, Polyspace does not ignore this defect.

#### Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <threads.h>
int thrd_func(void);
```

```
void initialize() {
    thrd_t thr;
    int n = 1;
    if( thrd_success!= thrd_create(&thr,thrd_func,&n) ){
        // handle error

    }
}
```

### Unprotected dynamic memory allocation

#### Issue

**Unprotected dynamic memory allocation** occurs when you do not check after dynamic memory allocation whether the memory allocation succeeded.

#### Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

#### Fix

Check the return value of `malloc`, `calloc`, or `realloc` for NULL before accessing the allocated memory location.

```
#DEFINE SIZE 8;

int *ptr = malloc(SIZE * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

#### Example - Unprotected Dynamic Memory Allocation Error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));
  *p = 2;  //Noncompliant
  /* Defect: p is not checked for NULL value */
  free(p);
}
/*Defect: p is not checked for NULL before deallocating*/
```

If the memory allocation fails, the function such as `calloc` returns `NULL` to `p`. Before accessing the memory through `p` or freeing `p`, the code does not check whether `p` is `NULL`. These operations might result in memory leaks.

#### Correction — Check for NULL Value

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>
```

```
void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));
   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;
   free(p);
 }
```

### Pointer overwritten during reallocation

#### Issue

**Pointer overwritten during reallocation** occurs when you overwrite the original pointer by the return value of `realloc()`. For instance:

```
p = realloc(p,SIZE);
```

#### Risk

The function `realloc()` returns a `NULL` value when memory allocation fails. In the preceding code, because you overwrite `p` by the return of `realloc()`, it becomes `NULL` when the reallocation operation fails. You lose the connection between the original memory block and `p`, resulting in a memory leak.

#### Fix

When reallocating pointers, preserve the original pointer. For instance, you might use a temporary variable to store the reallocated memory.

#### Example — Avoid Overwriting Original Pointer When Reallocating Memory

```
#include <stdlib.h>
//...
void foo (int* ptrI, size_t new_size)
{

  if (new_size == 0) {
    /* Handle error */
    return;
  }

  ptrI = (int*)realloc (ptrI, new_size);   //Noncompliant

  if (ptrI == NULL) {
    /* Handle error */
    return;
  }
}
```

Overwriting the pointer `ptrI` by the pointer returned by `realloc` destroys the association between `ptrI` and the original memory block. If `realloc` fails, such overwriting might cause a memory leak and data loss.

#### Correction — Store Reallocated Memory in Temporary Variable

When reallocating a pointer, use a temporary variable to hold the reallocated memory. Before assigning the temporary variable to `ptrI`, check it for `NULL` value to avoid memory leaks and data loss.

```
#include <stdlib.h>

void foo (int* ptrI, size_t new_size)
{
int* temp;
  if (new_size == 0) {
    /* Handle error */
    return;
  }

  temp = (int*)realloc (ptrI, new_size);

  if (temp == NULL) {
    /* Handle error */
    return;
  }else{
      ptrI = temp;
  }
}
```

## Check Information

**Group:** Rule 12. Error Handling (ERR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR33-C

# CERT C: Rule ERR34-C

Detect errors when converting a string to a number

## Description

### Rule Definition

*Detect errors when converting a string to a number.*

### Polyspace Implementation

The rule checker checks for **Unsafe conversion from string to numerical value**.

## Examples

### Unsafe conversion from string to numerical value

#### Issue

**Unsafe conversion from string to numerical value** detects conversions from strings to integer or floating-point values. If your conversion method does not include robust error handling, a defect is raised.

#### Risk

Converting a string to numerical value can cause data loss or misinterpretation. Without validation of the conversion or error handling, your program continues with invalid values.

#### Fix

- Add additional checks to validate the numerical value.
- Use a more robust string-to-numeric conversion function such as `strtol`, `strtoll`, `strtoul`, or `strtoull`.

#### Example - Conversion With `atoi`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char* argv1)
{
    int s = 0;
    if (demo_check_string_not_empty(argv1))
    {
        s = atoi(argv1);  //Noncompliant
    }
    return s;
}
```

In this example, `argv1` is converted to an integer with `atoi`. `atoi` does not provide errors for an invalid integer string. The conversion can fail unexpectedly.

**Correction — Use `strtol` instead**

One possible correction is to use `strtol` to validate the input string and the converted integer.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char *argv1)
{
    char *c_str = argv1;
    char *end;
    long sl;

    if (demo_check_string_not_empty(c_str))
    {
        errno = 0; /* set errno for error check */
        sl = strtol(c_str, &end, 10);
        if (end == c_str)
        {
            (void)fprintf(stderr, "%s: not a decimal number\n", c_str);
        }
        else if ('\0' != *end)
        {
            (void)fprintf(stderr, "%s: extra characters: %s\n", c_str, end);
        }
        else if ((LONG_MIN == sl || LONG_MAX == sl) && ERANGE == errno)
        {
            (void)fprintf(stderr, "%s out of range of type long\n", c_str);
        }
        else if (sl > INT_MAX)
        {
            (void)fprintf(stderr, "%ld greater than INT_MAX\n", sl);
        }
        else if (sl < INT_MIN)
        {
            (void)fprintf(stderr, "%ld less than INT_MIN\n", sl);
        }
        else
        {
            return (int)sl;
        }
    }
    return 0;
}
```

### Check Information

**Group:** Rule 12. Error Handling (ERR)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

ERR34-C

# CERT C: Rule CON30-C

Clean up thread-specific storage

## Description

### Rule Definition

*Clean up thread-specific storage.*

### Polyspace Implementation

The rule checker checks for **Thread-specific memory leak**.

## Examples

### Thread-specific memory leak

#### Issue

**Thread-specific memory leak** occurs when you do not free thread-specific dynamically allocated memory before the end of a thread.

To create thread-specific storage, you generally do these steps:

1. You create a key for thread-specific storage.
2. You create the threads.
3. In each thread, you allocate storage dynamically and then associate the key with this storage.

   After the association, you can read the stored data later using the key.
4. Before the end of the thread, you free the thread-specific memory using the key.

The checker flags execution paths in the thread where the last step is missing.

The checker works on these families of functions:

- `tss_get` and `tss_set` (C11)
- `pthread_getspecific` and `pthread_setspecific` (POSIX)

#### Risk

The data stored in the memory is available to other processes even after the threads end (memory leak). Besides security vulnerabilities, memory leaks can shrink the amount of available memory and reduce performance.

#### Fix

Free dynamically allocated memory before the end of a thread.

You can explicitly free dynamically allocated memory with functions such as `free`.

Alternatively, when you create a key, you can associate a destructor function with the key. The destructor function is called with the key value as argument at the end of a thread. In the body of the destructor function, you can free any memory associated with the key. If you use this method, Bug Finder still flags a defect. Ignore this defect with appropriate comments. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example - Memory Not Freed at End of Thread**

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;

  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */ //Noncompliant
  }
  print_data();
  return 0; //Noncompliant
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }
  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

In this example, the start function of each thread `func` calls two functions:

- `add_data`: This function allocates storage dynamically and associates the storage with a key using the `tss_set` function.
- `print_data`: This function reads the stored data using the `tss_get` function.

At the points where `func` returns, the dynamically allocated storage has not been freed.

**Correction — Free Dynamically Allocated Memory Explicitly**

One possible correction is to free dynamically allocated memory explicitly before leaving the start function of a thread. See the highlighted change in the corrected version.

In this corrected version, a defect still appears on the `return` statement in the error handling section of `func`. The defect cannot occur in practice because the error handling section is entered only if dynamic memory allocation fails. Ignore this remaining defect with appropriate comments. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;
  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */ //Noncompliant
  }
  print_data();
  free(tss_get(key));
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }

  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

## Check Information

**Group:** Rule 14. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**

“Check for and Review Coding Standard Violations”

**External Websites**

CON30-C

# CERT C: Rule CON31-C

Do not destroy a mutex while it is locked

## Description

### Rule Definition

*Do not destroy a mutex while it is locked.*

### Polyspace Implementation

The rule checker checks for **Destruction of locked mutex**.

### Extend Checker

For the rule checker to detect issues, your code must use one of the concurrency primitives recognized by Polyspace for thread creation and shared variable protection. Otherwise, you must explicitly specify tasks, interrupts, critical sections, and other multitasking options in your project configuration. See "Multitasking".

## Examples

### Destruction of locked mutex

#### Issue

**Destruction of locked mutex** occurs when a task destroys a mutex after it is locked (and before it is unlocked). The locking and destruction can happen in the same task or different tasks.

#### Risk

A mutex is locked to protect shared variables from concurrent access. If a mutex is destroyed in the locked state, the protection does not apply.

#### Fix

To fix this defect, destroy the mutex only after you unlock it. It is a good design practice to:

- Initialize a mutex *before* creating the threads where you use the mutex.
- Destroy a mutex *after* joining the threads that you created.

On the **Result Details** pane, you see two events, the locking and destruction of the mutex, and the tasks that initiated the events. To navigate to the corresponding line in your source code, click the event.

#### Example - Locking and Destruction in Different Tasks

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
  pthread_mutex_unlock (&lock3);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_destroy (&lock3); //Noncompliant
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

In this example, after task `t0` locks the mutex `lock3`, task `t1` can destroy it. The destruction occurs if the following events happen in sequence:

1. `t0` acquires `lock3`.
2. `t0` releases `lock2`.
3. `t0` releases `lock1`.
4. `t1` acquires the lock `lock1` released by `t0`.
5. `t1` acquires the lock `lock2` released by `t0`.
6. `t1` destroys `lock3`.

For simplicity, this example uses a mix of automatic and manual concurrency detection. The tasks `t0` and `t1` are manually specified as entry points by using the option `Tasks (-entry-points)`. The critical sections are implemented through primitives `pthread_mutex_lock` and `pthread_mutex_unlock` that the software detects automatically. In practice, for entry point specification (thread creation), you will use primitives such as `pthread_create`. The next example shows how the defect can appear when you use `pthread_create`.

**Correction — Place Lock-Unlock Pair Together in Same Critical Section as Destruction**

The locking and destruction of `lock3` occurs inside the critical section imposed by `lock1` and `lock2`, but the unlocking occurs outside. One possible correction is to place the lock-unlock pair in the same critical section as the destruction of the mutex. Use one of these critical sections:

- Critical section imposed by `lock1` alone.
- Critical section imposed by `lock1` and `lock2`.

In this corrected code, the lock-unlock pair and the destruction is placed in the critical section imposed by `lock1` and `lock2`. When `t0` acquires `lock1` and `lock2`, `t1` has to wait for their release before it executes the instruction `pthread_mutex_destroy (&lock3);`. Therefore, `t1` cannot destroy mutex `lock3` in the locked state.

```
#include <pthread.h>
```

```
pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);

  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);

  pthread_mutex_destroy (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

**Example - Locking and Destruction in Start Routine of Thread**

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_exit((void*) 0);
}

void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_destroy(&lock); //Noncompliant
    pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Thread that initializes mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use mutex for atomic operation*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   pthread_exit(NULL);
}
```

In this example, four threads are created. The threads are assigned different actions.

- The first thread `callThd[0]` initializes the mutex `lock`.
- The second and third threads, `callThd[1]` and `callThd[2]`, perform an atomic operation protected by the mutex `lock`.
- The fourth thread `callThd[3]` destroys the mutex `lock`.

The threads can interrupt each other. Therefore, immediately after the second or third thread locks the mutex, the fourth thread can destroy it.

**Correction — Initialize and Destroy Mutex Outside Start Routine**

One possible correction is to initialize and destroy the mutex in the `main` function outside the start routine of the threads. The threads perform only the atomic operation. You need two fewer threads because the mutex initialization and destruction threads are not required.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 2
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_work(void *arg) {
   pthread_mutex_lock (&lock);
   atomic_operation();
   pthread_mutex_unlock (&lock);
   pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize mutex */
   pthread_mutex_init(&lock, NULL);

   for(i=0; i<NUMTHREADS; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy mutex */
   pthread_mutex_destroy(&lock);

   pthread_exit(NULL);
}
```

#### Correction — Use A Second Mutex To Protect Lock-Unlock Pair and Destruction

Another possible correction is to use a second mutex and protect the lock-unlock pair from the destruction. This corrected code uses the mutex `lock2` to achieve this protection. The second mutex is initialized in the `main` function outside the start routine of the threads.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
pthread_mutex_t lock2;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}

void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_destroy(&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}


int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize second mutex */
   pthread_mutex_init(&lock2, NULL);

   /* Thread that initializes first mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use first mutex for atomic operation */
   /* The threads use second mutex to protect first from destruction in locked state*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys first mutex */
   /* The thread uses the second mutex to prevent destruction of locked mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);


   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy second mutex */
   pthread_mutex_destroy(&lock2);

   pthread_exit(NULL);
}
```

## Check Information

**Group:** Rule 14. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CON31-C

# CERT C: Rule CON32-C

Prevent data races when accessing bit fields from multiple threads

## Description

### Rule Definition

*Prevent data races when accessing bit fields from multiple threads.*

### Polyspace Implementation

The rule checker checks for **Data race on adjacent bit fields**.

### Extend Checker

For the rule checker to detect issues, your code must use one of the concurrency primitives recognized by Polyspace for thread creation and shared variable protection. Otherwise, you must explicitly specify tasks, interrupts, critical sections, and other multitasking options in your project configuration. See “Multitasking”.

## Examples

### Data race on adjacent bit fields

#### Issue

Data race on adjacent bit fields occurs when both of these conditions are true:

- Multiple tasks perform unprotected operations on bit fields that are part of the same structure.

  For instance, a task operates on field `errorFlag1` and another task on field `errorFlag2` in a variable of this type:

  ```
  struct errorFlags {
     unsigned int errorFlag1 : 1;
     unsigned int errorFlag2 : 1;
     //...
  };
  ```

  Suppose that the operations are not atomic with respect to each other. In other words, you have not implemented protection mechanisms to ensure that one operation is completed before another operation begins.
- At least one of the unprotected operations is a write operation.

To find this defect, before analysis, you must specify the multitasking options. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

#### Risk

Adjacent bit fields that are part of the same structure might be stored in one byte in the same memory location. Read or write operations on all variables including bit fields occur one byte or word at a time. To modify only specific bits in a byte, steps similar to these steps occur in sequence:

1. The byte is loaded into RAM.
2. A mask is created so that only specific bits are modified to the intended value and the remaining bits remain unchanged.
3. A bitwise OR operation is performed between the copy of the byte in RAM and the mask.
4. The byte with specific bits modified is copied back from RAM.

When you access two different bit fields, these four steps have to be performed for each bit field. If the accesses are not protected, all four steps for one bit field might not be completed before the four steps for the other bit field begin. As a result, the modification of one bit field might undo the modification of an adjacent bit field. For instance, in the preceding example, the modification of `errorFlag1` and `errorFlag2` can occur in the following sequence.

Steps 1,2 and 5 relate to modification of `errorFlag1` and while steps 3,4 and 6 relate to that of `errorFlag2`.

1. The byte with both `errorFlag1` and `errorFlag2` unmodified is copied into RAM, for purposes of modifying `errorFlag1`.
2. A mask that modifies only `errorFlag1` is bitwise OR-ed with this copy.
3. The byte containing both `errorFlag1` and `errorFlag2` unmodified is copied into RAM a second time, for purposes of modifying `errorFlag2`.
4. A mask that modifies only `errorFlag2` is bitwise OR-ed with this second copy.
5. The version with `errorFlag1` modified is copied back. This version has `errorFlag2` unmodified.
6. The version with `errorFlag2` modified is copied back. This version has `errorFlag1` unmodified and overwrites the previous modification.

### Fix

To fix this defect, protect the operations on bit fields that are part of the same structure using critical sections, temporal exclusion, or another means. See "Protections for Shared Variables in Multitasking Code".

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon.

### Example - Unprotected Operation on Global Variable from Multiple Tasks

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12; //Noncompliant
```

```
void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

In this example, `task1` and `task2` access different bit fields `IOFlag` and `SetupFlag`, which belong to the same structured variable `InterruptConfigbitsProc12`.

To emulate multitasking behavior, specify the options listed in this table.

| Option | Specification |
|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ |
| **Tasks** on page 2-121 | `task1`<br>`task2` |

At the command-line, use:

```
 polyspace-bug-finder
   -entry-points task1,task2
```

#### Correction – Use Critical Sections

One possible correction is to wrap the bit field access in a critical section. A critical section lies between a call to a lock function and an unlock function. In this correction, the critical section lies between the calls to functions `begin_critical_section` and `end_critical_section`.

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void begin_critical_section(void);
void end_critical_section(void);

void task1 (void) {
    begin_critical_section();
    InterruptConfigbitsProc12.IOFlag = 0;
    end_critical_section();
}
```

```
void task2 (void) {
    begin_critical_section();
    InterruptConfigbitsProc12.SetupFlag = 0;
    end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the options listed in this table.

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ | |
| **Tasks** on page 2-121 | `task1`<br>`task2` | |
| **Critical section details** on page 2-133 | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

At the command-line, use:

```
 polyspace-bug-finder
   -entry-points task1,task2
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

#### Correction – Avoid Bit Fields

If you do not have memory constraints, use the `char` data type instead of bit fields. The `char` variables in a structure occupy at least one byte and do not have the thread-safety issues that come from bit manipulations in a byte-sized operation. Data races do not result from unprotected operations on different `char` variables that are part of the same structure.

```
typedef struct
{
   unsigned char IOFlag;
   unsigned char InterruptFlag;
   unsigned char Register1Flag;
   unsigned char SignFlag;
   unsigned char SetupFlag;
   unsigned char Register2Flag;
   unsigned char ProcessorFlag;
   unsigned char GeneralFlag;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

Though the checker does not flag this correction, do not use this correction for C99 or earlier. Only from C11 and later does the C Standard mandate that distinct `char` variables cannot be accessed using the same word.

#### Correction – Insert Bit Field of Size 0

You can enter a non-bit field member or an unnamed bit field member of size 0 between two adjacent bit fields that might be accessed concurrently. A non-bit field member or size 0 bit-field member ensures that the subsequent bit field starts from a new memory location. In this corrected example, the size 0 bit-field member ensures that `IOFlag` and `SetupFlag` are stored in distinct memory locations.

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int : 0;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

### Check Information

**Group:** Rule 14. Concurrency (CON)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"
"Analyze Multitasking Programs in Polyspace"
"Protections for Shared Variables in Multitasking Code"

#### External Websites

CON32-C

# CERT C: Rule CON33-C

Avoid race conditions when using library functions

## Description

### Rule Definition

*Avoid race conditions when using library functions.*

### Polyspace Implementation

The rule checker checks for **Data race through standard library function call**.

### Extend Checker

For the rule checker to detect issues, your code must use one of the concurrency primitives recognized by Polyspace for thread creation and shared variable protection. Otherwise, you must explicitly specify tasks, interrupts, critical sections, and other multitasking options in your project configuration. See “Multitasking”.

## Examples

### Data race through standard library function call

#### Issue

**Data race through standard library function call** occurs when:

- Multiple tasks call the same standard library function.

  For instance, multiple tasks call the `strerror` function.
- The calls are not protected using a common protection.

  For instance, the calls are not protected by the same critical section.

Functions flagged by this defect are not guaranteed to be reentrant. A function is reentrant if it can be interrupted and safely called again before its previous invocation completes execution. If a function is not reentrant, multiple tasks calling the function without protection can cause concurrency issues. For the list of functions that are flagged, see CON33-C: Avoid race conditions when using library functions.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

#### Risk

The functions flagged by this defect are nonreentrant because their implementations can use global or static variables. When multiple tasks call the function without protection, the function call from one task can interfere with the call from another task. The two invocations of the function can concurrently access the global or static variables and cause unpredictable results.

The calls can also cause more serious security vulnerabilities, such as abnormal termination, denial-of-service attack, and data integrity violations.

**Fix**

To fix this defect, do one of the following:

- Use a reentrant version of the standard library function if it exists.

  For instance, instead of `strerror()`, use `strerror_r()` or `strerror_s()`. For alternatives to functions flagged by this defect, see the documentation for CON33-C.
- Protect the function calls using common critical sections or temporal exclusion.

  See `Critical section details (-critical-section-begin -critical-section-end)` and `Temporally exclusive tasks (-temporal-exclusions-file)`.

  To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon. For an example, see below.

**Example - Unprotected Call to Standard Library Function from Multiple Tasks**

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno); //Noncompliant
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
    end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Specification | |
|---|---|---|
| Configure multitasking manually | ☑ | |
| Tasks (-entry-points) | task1<br>task2<br>task3 | |
| Critical section details (-critical-section-begin -critical-section-end) | **Starting routine** | **Ending routine** |
| | begin_critical_section | end_critical_section |

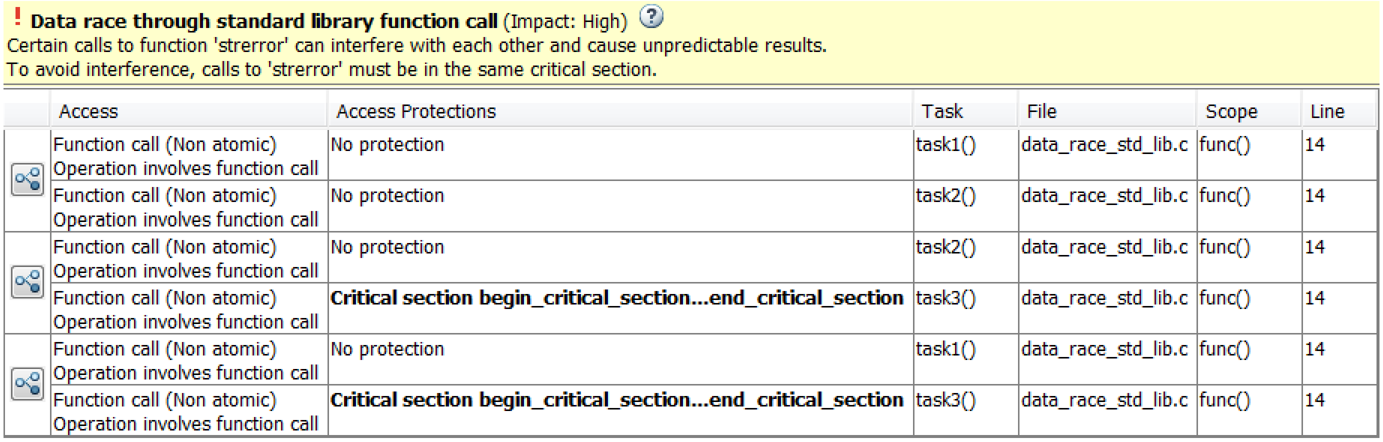
On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2,task3
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

In this example, the tasks, `task1`, `task2` and `task3`, call the function `func`. `func` calls the nonreentrant standard library function, `strerror`.

Though `task3` calls `func` inside a critical section, other tasks do not use the same critical section. Operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

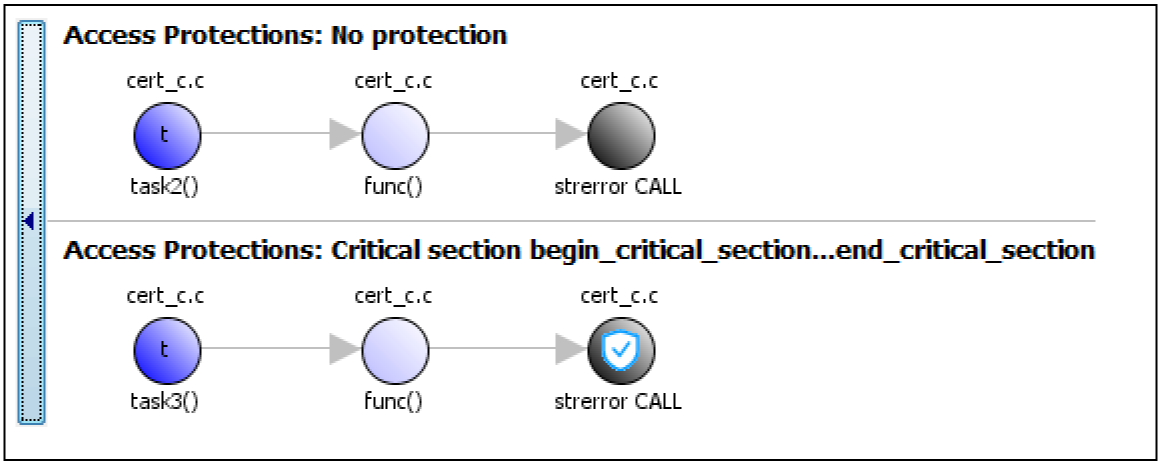
These three tasks are calling a nonreentrant standard library function without common protection. In your result details, you see each pair of conflicting function calls.



! **Data race through standard library function call** (Impact: High) ?
Certain calls to function 'strerror' can interfere with each other and cause unpredictable results.
To avoid interference, calls to 'strerror' must be in the same critical section.

| | Access | Access Protections | Task | File | Scope | Line |
|---|---|---|---|---|---|---|
| | Function call (Non atomic)<br>Operation involves function call | No protection | task1() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task2() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task2() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | **Critical section begin_critical_section...end_critical_section** | task3() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task1() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | **Critical section begin_critical_section...end_critical_section** | task3() | data_race_std_lib.c | func() | 14 |

If you click the icon, you see the function call sequence starting from the entry point to the standard library function call. You also see that the call starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.


Access Protections: No protection
cert_c.c
cert_c.c
cert_c.c
t
task2()
func()
strerror CALL
Access Protections: Critical section begin_critical_section...end_critical_section
cert_c.c
cert_c.c
cert_c.c
t
task3()
func()
strerror CALL

**Correction — Use Reentrant Version of Standard Library Function**

One possible correction is to use a reentrant version of the standard library function `strerror`. You can use the POSIX version `strerror_r` which has the same functionality but also guarantees thread-safety.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);
enum { BUFFERSIZE = 64 };

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char errmsg[BUFFERSIZE];
    if (strerror_r(errno, errmsg, BUFFERSIZE) != 0) {
      /* Handle error */
    }
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}
```

```
void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

**Correction — Place Function Call in Critical Section**

One possible correction is to place the call to `strerror` in critical section. You can implement the critical section in multiple ways.

For instance, you can place the call to the intermediate function `func` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `func` and therefore the calls to `strerror` from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call `func` between calls to `begin_critical_section` and `end_critical_section`.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    begin_critical_section();
    func(fptr1);
    end_critical_section();
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     begin_critical_section();
     func(fptr2);
     end_critical_section();
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

#### Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

| Option | Value |
|---|---|
| `Temporally exclusive tasks (-temporal-exclusions-file)` | `task1 task2 task3` |

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

## Check Information

**Group:** Rule 14. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CON33-C

# CERT C: Rule CON34-C

Declare objects shared between threads with appropriate storage durations

## Description

### Rule Definition

*Declare objects shared between threads with appropriate storage durations.*

### Polyspace Implementation

The rule checker checks for **Automatic or thread local variable escaping from a C11 thread**.

## Examples

### Automatic or thread local variable escaping from a C11 thread

#### Issue

**Automatic or thread local variable escaping from a C11 thread** occurs when an automatic or thread local variable is passed by address from one C11 thread to another without ensuring that the variable stays alive through the duration of the latter thread.

#### Risk

An automatic or thread local variable is allocated on the stack at the beginning of a thread and its lifetime extends till the end of the thread. The variable is not guaranteed to be alive when a different thread accesses it.

For instance, consider the start function of a C11 thread with these lines:

```
int start_thread(thrd_t *tid) {
   int aVar = 0;
   if(thrd_success != thrd_create(tid, start_thread_child, &aVar) {
     //...
   }
}
```

The `thrd_create` function creates a child thread with start function `start_thread_child` and passes the address of the automatic variable `aVar`to this function. When this child thread accesses `aVar`, the parent thread might have completed execution and `aVar` is no longer on the stack. The access might result in reading unpredictable values.

#### Fix

When you pass a variable from one thread to another, make sure that the variable lifetime matches or exceeds the lifetime of both threads. You can achieve this synchronization in one of these ways:

- Declare the variable `static` so that it does not go out of stack when the current thread completes execution.
- Dynamically allocate the storage for the variable so that it is allocated on the heap instead of the stack and must be explicitly deallocated. Make sure that the deallocation happens after both threads complete execution.

These solutions require you to create a variable in nonlocal memory. Instead, you can use other solutions such as the `shared` keyword available with OpenMP's threading interface that allows you to safely share local variables across threads.

**Example – Automatic or Thread-Local Variable Escaping Thread**

```
#include <threads.h>
#include <stdio.h>

int create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return 0;
}

void create_parent_thread(thrd_t *tid, int *parentPtr) {
   if (thrd_success != thrd_create(tid, create_child_thread, parentPtr)) { //Noncompliant
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  int parentVal = 1;

  create_parent_thread(&tid, &parentVal);


  if (thrd_success != thrd_join(tid, NULL)) {
    /* Handle error */
  }
  return 0;
}
```

In this example, the value `parentVal` is local to the parent thread that starts in `main` and continues into the function `create_parent_thread`. However, in the body of `create_parent_thread`, the address of this local variable is passed to a child thread (the thread with start routine `create_child_thread`). The parent thread might have completed execution and the variable `parentVal` might have gone out of scope when the child thread accesses this variable.

The same issue appears if the variable is declared as thread-local, for instance with the C11 keyword `_Thread_local` (or `thread_local`):

```
_Thread_local int parentVal = 1;
```

**Correction – Use Static Variables**

One possible correction is to declare the variable `parentVal` as `static` so that the variable is on the stack for the entire duration of the program.

```
#include <threads.h>
#include <stdio.h>

int create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return 0;
}
```

```
void create_parent_thread(thrd_t *tid, int *parentPtr) {
   if (thrd_success != thrd_create(tid, create_child_thread, parentPtr)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  static int parentVal = 1;

  create_parent_thread(&tid, &parentVal);


  if (thrd_success != thrd_join(tid, NULL)) {
    /* Handle error */
  }
  return 0;
}
```

**Correction – Use Dynamic Memory Allocation**

One possible correction is to dynamically allocate storage for variables to be shared across threads and explicitly free the storage after the variable is no longer required.

```
#include <threads.h>
#include <stdio.h>

int create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return 0;
}

void create_parent_thread(thrd_t *tid, int *parentPtr) {
   if (thrd_success != thrd_create(tid, create_child_thread, parentPtr)) {
    /* Handle error */
  }
}

int main(void) {
  thrd_t tid;
  int parentVal = 1;
  int parentPtr = (int*) malloc(sizeof(int));

  if(!parentPtr) {
      create_parent_thread(&tid, parentPtr);


      if (thrd_success != thrd_join(tid, NULL)) {
        /* Handle error */
      }
      free(parentPtr);
  }
  return 0;
}
```

## Check Information

**Group:** Rule 14. Concurrency (CON)

## Version History

**Introduced in R2020a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CON34-C

# CERT C: Rule CON35-C

Avoid deadlock by locking in a predefined order

## Description

### Rule Definition

*Avoid deadlock by locking in a predefined order.*

### Polyspace Implementation

The rule checker checks for **Deadlock**.

### Extend Checker

For the rule checker to detect issues, your code must use one of the concurrency primitives recognized by Polyspace for thread creation and shared variable protection. Otherwise, you must explicitly specify tasks, interrupts, critical sections, and other multitasking options in your project configuration. See “Multitasking”.

You can also extend this checker to detect synchronization issues by mapping your multithreading functions to their known POSIX equivalents. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.

## Examples

### Deadlock

#### Issue

**Deadlock** occurs when multiple tasks are stuck in their critical sections (CS) because:

- Each CS waits for another CS to end.
- The critical sections (CS) form a closed cycle. For example:
  - CS #1 waits for CS #2 to end, and CS #2 waits for CS #1 to end.
  - CS #1 waits for CS #2 to end, CS #2 waits for CS #3 to end and CS #3 waits for CS #1 to end.

Polyspace expects critical sections of code to follow a specific format. A critical section lies between a call to a lock function and a call to an unlock function. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function. Both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

#### Risk

Each task waits for a critical section in another task to end and is unable to proceed. The program can freeze indefinitely.

#### Fix

The fix depends on the root cause of the defect. You can try to break the cyclic order between the tasks in one of these ways:

- Write down all critical sections involved in the deadlock in a certain sequence. Whenever you call the lock functions of the critical sections within a task, respect the order in that sequence. See an example below.
- If one of the critical sections involved in a deadlock occurs in an interrupt, try to disable all interrupts during critical sections in all tasks. See `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Reviewing this defect is an opportunity to check if all operations in your critical section are really meant to be executed as an atomic block. It is a good practice to keep critical sections at a bare minimum.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Deadlock with Two Tasks

```
void task1(void);
void task2(void);

int var;
void perform_task_cycle(void) {
 var++;
}

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2(); //Noncompliant
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_2();
    begin_critical_section_1();
    perform_task_cycle();
    end_critical_section_1();
    end_critical_section_2();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** | ☑ | |
| **Entry points** | task1<br>task2 | |
| **Critical section details** | **Starting routine** | **Ending routine** |
| | begin_critical_section_1 | end_critical_section_1 |
| | begin_critical_section_2 | end_critical_section_2 |

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `begin_critical_section_1`.
2. `task2` calls `begin_critical_section_2`.
3. `task1` reaches the instruction `begin_critical_section_2();`. Since `task2` has already called `begin_critical_section_2`, `task1` waits for `task2` to call `end_critical_section_2`.
4. `task2` reaches the instruction `begin_critical_section_1();`. Since `task1` has already called `begin_critical_section_1`, `task2` waits for `task1` to call `end_critical_section_1`.

#### Correction-Follow Same Locking Sequence in Both Tasks

One possible correction is to follow the same sequence of calls to lock and unlock functions in both `task1` and `task2`.

```
void task1(void);
void task2(void);
void perform_task_cycle(void);

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}
```

**Example - Deadlock with More Than Two Tasks**

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);


void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2(); //Noncompliant
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock3();
    lock1();
    performTaskCycle();
    unlock1();
    unlock3();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** | ☑ | |
| **Entry points** | `task1`<br>`task2`<br>`task3` | |
| **Critical section details** | **Starting routine** | **Ending routine** |
| | `lock1` | `unlock1` |
| | `lock2` | `unlock2` |
| | `lock3` | `unlock3` |

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `lock1`.
2. `task2` calls `lock2`.
3. `task3` calls `lock3`.
4. `task1` reaches the instruction `lock2();`. Since `task2` has already called `lock2`, `task1` waits for call to `unlock2`.
5. `task2` reaches the instruction `lock3();`. Since `task3` has already called `lock3`, `task2` waits for call to `unlock3`.
6. `task3` reaches the instruction `lock1();`. Since `task1` has already called `lock1`, `task3` waits for call to `unlock1`.

#### Correction — Break Cyclic Order

To break the cyclic order between critical sections, note every lock function in your code in a certain sequence, for example:

1. `lock1`
2. `lock2`
3. `lock3`

If you use more than one lock function in a task, use them in the order in which they appear in the sequence. For example, you can use `lock1` followed by `lock2` but not `lock2` followed by `lock1`.

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);

void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock1();
    lock3();
    performTaskCycle();
    unlock3();
    unlock1();
 }
}
```

## Check Information

**Group:** Rule 14. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**
CON35-C

# CERT C: Rule CON36-C

Wrap functions that can spuriously wake up in a loop

## Description

### Rule Definition

*Wrap functions that can spuriously wake up in a loop.*

### Polyspace Implementation

The rule checker checks for **Function that can spuriously wake up not wrapped in loop**.

## Examples

### Function that can spuriously wake up not wrapped in loop

#### Issue

**Function that can spuriously wake up not wrapped in loop** occurs when the following wait-on-condition functions are called from outside a loop:

- C functions:
  - `cnd_wait()`
  - `cnd_timedwait()`
- POSIX functions:
  - `pthread_cond_wait()`
  - `pthread_cond_timedwait()`
- C++ `std::condition_variable` and `std::condition_variable_any` class member functions:
  - `wait()`
  - `wait_until()`
  - `wait_for()`

Wait-on-condition functions pause the execution of the calling thread when a specified condition is met. The thread wakes up and resumes once another thread notifies it with `cnd_broadcast()` or an equivalent function. The wake-up notification can be spurious or malicious.

#### Risk

If a thread receives a spurious wake-up notification and the condition of the wait-on-condition function is not checked, the thread can wake up prematurely. The wake-up can cause unexpected control flow, indefinite blocking of other threads, or denial of service.

#### Fix

Wrap wait-on-condition functions that can wake up spuriously in a loop. The loop checks the wake-up condition after a possible spurious wake-up notification.

**Example - `cnd_wait()` Not Wrapped in Loop**

```
#include <stdio.h>
#include <stddef.h>
#include <threads.h>

#define THRESHOLD 100

static mtx_t lock;
static cnd_t cond;

void func(int input)
{
    if (thrd_success != mtx_lock(&lock)) {
        /* Handle error */
    }
    /* test condition to pause thread */
    if (input > THRESHOLD) {
        if (thrd_success != cnd_wait(&cond, &lock)) { //Noncompliant
            /* Handle error */
        }
    }
    /* Proceed if condition to pause does not hold */


    if (thrd_success != mtx_unlock(&lock)) {
        /* Handle error */
    }
}
```

In this example, the thread uses `cnd_wait()` to pause execution when `input` is greater than `THRESHOLD`. The paused thread can resume if another thread uses `cnd_broadcast()`, which notifies all the threads. This notification causes the thread to wake up even if the pause condition is still true.

**Correction — Wrap `cnd_wait()` in a `while` Loop**

One possible correction is to wrap `cnd_wait()` in a `while` loop. The loop checks the pause condition after the thread receives a possible spurious wake-up notification.

```
#include <stdio.h>
#include <stddef.h>
#include <threads.h>

#define THRESHOLD 100

static mtx_t lock;
static cnd_t cond;

void func(int input)
{
    if (thrd_success != mtx_lock(&lock)) {
        /* Handle error */
    }
    /* test condition to pause thread */
    while (input > THRESHOLD) {
        if (thrd_success != cnd_wait(&cond, &lock)) {
            /* Handle error */
        }
    }
    /* Proceed if condition to pause does not hold */


    if (thrd_success != mtx_unlock(&lock)) {
        /* Handle error */
    }
}
```

## Check Information

**Group:** Rule 14. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CON36-C

# CERT C: Rule CON37-C

Do not call signal() in a multithreaded program

## Description

### Rule Definition

*Do not call signal() in a multithreaded program.*

### Polyspace Implementation

The rule checker checks for **Signal call in multithreaded program**.

## Examples

### Signal call in multithreaded program

#### Issue

**Signal call in multithreaded program** occurs when you use the `signal()` function in a program with multiple threads.

#### Risk

According to the C11 standard (Section 7.14.1.1), use of the `signal()` function in a multithreaded program is undefined behavior.

#### Fix

Depending on your intent, use other ways to perform an asynchronous action on a specific thread.

#### Example - Use of `signal()` Function to Terminate Loop in Thread

```
#include <signal.h>
#include <stddef.h>
#include <threads.h>

volatile sig_atomic_t flag = 0;

void handler(int signum) {
  flag = 1;
}

/* Runs until user sends SIGUSR1 */
int func(void *data) {
  while (!flag) {
    /* ... */
  }
  return 0;
}

int main(void) {
  signal(SIGINT, handler); /* Undefined behavior */ //Noncompliant
  thrd_t tid;
  if (thrd_success != thrd_create(&tid, func, NULL)) {
    /* Handle error */
  }
  /* ... */
  return 0;
}
```

In this example, the `signal` function is used to terminate a `while` loop in the thread created with `thrd_create`.

### Correction — Use `atomic_bool` Variable to Terminate Loop

One possible correction is to use an `atomic_bool` variable that multiple threads can access. In the corrected example, the child thread evaluates this variable before every loop iteration. After completing the program, you can modify this variable so that the child thread exits the loop.

```
#include <stdatomic.h>
#include <stdbool.h>
#include <stddef.h>
#include <threads.h>

atomic_bool flag = ATOMIC_VAR_INIT(false);

int func(void *data) {
  while (!flag) {
    /* ... */
  }
  return 0;
}

int main(void) {
  thrd_t tid;

  if (thrd_success != thrd_create(&tid, func, NULL)) {
    /* Handle error */
  }
  /* ... */
  /* Set flag when done */
  flag = true;

  return 0;
}
```

## Check Information

**Group:** Rule 14. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**
“Check for and Review Coding Standard Violations”

**External Websites**
CON37-C

# CERT C: Rule CON38-C

Preserve thread safety and liveness when using condition variables

## Description

### Rule Definition

*Preserve thread safety and liveness when using condition variables.*

### Polyspace Implementation

The rule checker checks for **Multiple threads waiting on same condition variable**.

## Examples

### Multiple threads waiting on same condition variable

#### Issue

This issue occurs when you use `cnd_signal` family functions to wake up one of at least two threads that are concurrently waiting on the same condition variable. For threads with the same priority level, `cnd_signal` family functions cause the thread scheduler to arbitrarily wake up on of the threads waiting on the condition variable that you signal with the `cnd_signal` family function.

The checkers flags the `cnd_signal` family function call. See the **Event** column in the **Results Details** pane to view the threads waiting on the same condition variable.

#### Risk

The thread that is woken up with a `cnd_signal` family function usually tests for a condition predicate. While the condition predicate is false, the thread waits again on the condition variable until it is woken up by another thread that signals the condition variable. It is possible that the program ends up in a state where no thread is available to signal the condition variable, which results in indefinite blocking.

#### Fix

Use `cnd_broadcast` family functions instead to wake all threads waiting on the condition variable, or use a different condition variable for each thread.

#### Example - Use of `cnd_signal` to Wake Up One of Many Threads Waiting on Condition Variable

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <threads.h>

typedef int thrd_return_t;

static void fatal_error(void)
{
    exit(1);
}
```

```
enum { NTHREADS = 5 };

mtx_t mutex;
cnd_t cond;

thrd_return_t next_step(void* t)
{
    static size_t current_step = 0;
    size_t my_step = *(size_t*)t;

    if (thrd_success != mtx_lock(&mutex)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu has the lock\n", my_step);
    while (current_step != my_step) {
        printf("Thread %zu is sleeping...\n", my_step);
        if (thrd_success !=
            cnd_wait(&cond, &mutex)) {
            /* Handle error */
            fatal_error();
        }
        printf("Thread %zu woke up\n", my_step);
    }
    /* Do processing ... */
    printf("Thread %zu is processing...\n", my_step);
    current_step++;

    /* Signal a waiting task */
    if (thrd_success !=
        cnd_signal(&cond)) { //Noncompliant
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu is exiting...\n", my_step);

    if (thrd_success != mtx_unlock(&mutex)) {
        /* Handle error */
        fatal_error();
    }
    return (thrd_return_t)0;
}

int main(void)
{
    thrd_t threads[NTHREADS];
    size_t step[NTHREADS];

    if (thrd_success != mtx_init(&mutex, mtx_plain)) {
        /* Handle error */
        fatal_error();
    }
    if (thrd_success != cnd_init(&cond)) {
        /* Handle error */
        fatal_error();
    }
    /* Create threads */
    for (size_t i = 0; i < NTHREADS; ++i) {
        step[i] = i;
        if (thrd_success != thrd_create(&threads[i],
                                        next_step,
                                        &step[i])) {
            /* Handle error */
            fatal_error();
        }
    }
    /* Wait for all threads to complete */
    for (size_t i = NTHREADS; i != 0; --i) {
        if (thrd_success != thrd_join(threads[i - 1], NULL)) {
            /* Handle error */
            fatal_error();
        }
    }
    (void)mtx_destroy(&mutex);
    (void)cnd_destroy(&cond);
    return 0;
}
```

In this example, multiple threads are created and assigned step level. Each thread checks if its assigned step level matches the current step level (condition predicate). If the predicate is false, the thread goes back to waiting on the condition variable `cond`. The use of `cnd_signal` to signal the `cond` causes the thread scheduler to arbitrarily wake up one of the threads waiting on `cond`. This can result in indefinite blocking when the condition predicate of woken up thread is false and no other thread is available to signal `cond`.

**Correction — Use `cnd_broadcast` to Wake up All the Threads**

One possible correction is to use `cnd_broadcast` instead to signal `cond`. The function `cnd_signal` wakes up all the thread that are waiting on `cond`.

```
 #include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <threads.h>

typedef int thrd_return_t;

static void fatal_error(void)
{
    exit(1);
}

enum { NTHREADS = 5 };

mtx_t mutex;
cnd_t cond;

thrd_return_t next_step(void* t)
{
    static size_t current_step = 0;
    size_t my_step = *(size_t*)t;

    if (thrd_success != mtx_lock(&mutex)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu has the lock\n", my_step);
    while (current_step != my_step) {
        printf("Thread %zu is sleeping...\n", my_step);
        if (thrd_success !=
            cnd_wait(&cond, &mutex)) {
            /* Handle error */
            fatal_error();
        }
        printf("Thread %zu woke up\n", my_step);
    }
    /* Do processing ... */
    printf("Thread %zu is processing...\n", my_step);
    current_step++;

    /* Signal a waiting task */
    if (thrd_success !=
        cnd_broadcast(&cond)) {
        /* Handle error */
        fatal_error();
    }

    printf("Thread %zu is exiting...\n", my_step);

    if (thrd_success != mtx_unlock(&mutex)) {
        /* Handle error */
        fatal_error();
    }
    return (thrd_return_t)0;
}

int main_test_next_step(void)
{
    thrd_t threads[NTHREADS];
    size_t step[NTHREADS];

    if (thrd_success != mtx_init(&mutex, mtx_plain)) {
        /* Handle error */
        fatal_error();
    }
    if (thrd_success != cnd_init(&cond)) {
        /* Handle error */
        fatal_error();
    }
    /* Create threads */
    for (size_t i = 0; i < NTHREADS; ++i) {
        step[i] = i;
        if (thrd_success != thrd_create(&threads[i],
                                        next_step,
                                        &step[i])) {
            /* Handle error */
            fatal_error();
        }
    }
    /* Wait for all threads to complete */
    for (size_t i = NTHREADS; i != 0; --i) {
        if (thrd_success != thrd_join(threads[i - 1], NULL)) {
            /* Handle error */
            fatal_error();
        }
    }
    (void)mtx_destroy(&mutex);
    (void)cnd_destroy(&cond);
    return 0;
}
```

## Check Information

**Group:** Rule 14. Concurrency (CON)

# Version History

**Introduced in R2020a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CON38-C

# CERT C: Rule CON39-C

Do not join or detach a thread that was previously joined or detached

## Description

### Rule Definition

*Do not perform operations that can block while holding a lock.*

### Polyspace Implementation

The rule checker checks for **Join or detach of a joined or detached thread**.

## Examples

### Join or detach of a joined or detached thread

#### Issue

**Join or detach of a joined or detached thread** occurs when:

- A thread that is joined was previously joined or detached
- A thread that is detached was previously joined or detached.

The **Result Details** pane describes if the thread was previously joined or detached and also shows previous related events.

For instance, the issue occurs when a thread joined with `thrd_join` is then detached with `pthread_detach`:

```
thrd_t id;
//...
thrd_join(id, NULL);
thrd_detach(id);
```

Note that a thread is considered as joined only if a previous thread joining is successful. For instance, the thread is not considered as joined in the `if` branch here:

```
thrd_t t;
//...
if (thrd_success != thrd_join(t, 0)) {
    /* Thread not considered joined */
}
```

The analysis cannot detect cases where a joined thread detaches itself using, for instance, the `thrd_current()` function.

#### Risk

The C11 standard (clauses 7.26.5.3 and 7.26.5.6) states that a thread shall not be joined or detached once it was previously joined or detached. Violating these clauses of the standard results in undefined behavior.

#### Fix

Avoid joining a thread that was already joined or detached previously. Likewise, avoid detaching a thread that was already joined or detached.

#### Example – Joining Followed by Detaching of Thread

```
#include <stddef.h>
#include <threads.h>
#include <stdlib.h>


extern int thread_func(void *arg);

int main (void)
{
  thrd_t t;

  if (thrd_success != thrd_create (&t, thread_func, NULL)) {
    /* Handle error */
    return 0;
  }

  if (thrd_success != thrd_join (t, 0)) {
    /* Handle error */
    return 0;
  }

  if (thrd_success != thrd_detach (t)) {    //Noncompliant
    /* Handle error */
    return 0;
  }

  return 0;
}
```

In this example, the use of `thrd_detach` on a thread that was previously joined with `thrd_join` leads to undefined behavior.

To avoid compilation errors with this example, specify the C11 standard with the option `C standard version (-c-version)`.

#### Correction – Avoid Detaching a Joined Thread

Remove the `thrd_join` or `thrd_detach` statement.

```
#include <stddef.h>
#include <threads.h>
#include <stdlib.h>


extern int thread_func(void *arg);

int main (void)
{
  thrd_t t;
  if (thrd_success != thrd_create (&t, thread_func, NULL)) {
    /* Handle error */
    return 0;
  }

  if (thrd_success != thrd_join (t, 0)) {
    /* Handle error */
    return 0;
  }

  return 0;
}
```

### Example – Joining Thread Created in Detached State

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success != pthread_attr_init(&attr)) {
        return 0;
    }

    if(thread_success != pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED)) {
        return 0;
    }

    if(thread_success != pthread_create(&id, &attr, thread_func, NULL)) {
            return 0;
    }

    if(thread_success != pthread_join(id, NULL)) { //Noncompliant
            return 0;
    }

    return 0;
}
```

In this example, the thread attribute is assigned the state `PTHREAD_CREATE_DETACHED`. A thread created using this attribute is then joined.

### Correction – Create Threads as Joinable

One possible correction is to create a thread with thread attribute assigned to the state `PTHREAD_CREATE_JOINABLE` and then join the thread.

```
#include <stddef.h>
#include <pthread.h>
#define thread_success 0

extern void *thread_func(void *arg);


int main() {
    pthread_t id;
    pthread_attr_t attr;

    if(thread_success != pthread_attr_init(&attr)) {
        return 0;
    }

    if(thread_success != pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE)) {
        return 0;
    }

    if(thread_success != pthread_create(&id, &attr, thread_func, NULL)) {
            return 0;
    }

    if(thread_success != pthread_join(id, NULL)) {
            return 0;
    }

    return 0;
}
```

## Check Information

**Group:** Rule 14. Concurrency (CON)

# Version History

**Introduced in R2019b**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CON39-C

# CERT C: Rule CON40-C

Do not refer to an atomic variable twice in an expression

## Description

### Rule Definition

*Do not refer to an atomic variable twice in an expression.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Atomic variable accessed twice in an expression**.
- **Atomic load and store sequence not atomic**.

## Examples

### Atomic variable accessed twice in an expression

#### Issue

**Atomic variable accessed twice in an expression** occurs when C atomic types or C++ `std::atomic` class variables appear twice in an expression and there are:

- Two atomic read operations on the variable.
- An atomic read and a distinct atomic write operation on the variable.

The C standard defines certain operations on atomic variables that are thread safe and do not cause data race conditions. Unlike individual operations, a pair of operations on the same atomic variable in an expression is not thread safe.

#### Risk

A thread can modify the atomic variable between the pair of atomic operations, which can result in a data race condition.

#### Fix

Do not reference an atomic variable twice in the same expression.

#### Example - Referencing Atomic Variable Twice in an Expression

In this example, the global variable `n` is referenced twice in the return statement of `compute_sum()`. The value of `n` can change between the two distinct read operations. `compute_sum()` can return an incorrect value.

To see this violation, run Polyspace Bug Finder on this code and spcify `gnu4.9` as the compiler. See `Compiler (-compiler)`.

```
#include <stdatomic.h>

atomic_int n = ATOMIC_VAR_INIT(0);
```

```
int compute_sum(void)
{
    return n * (n + 1) / 2; //Noncompliant
}
```

#### Correction — Pass Variable as Function Argument

One possible correction is to pass the variable as a function argument `n`. The variable is copied to memory and the read operations on the copy guarantee that `compute_sum()` returns a correct result. If you pass a variable of type `int` instead of type `atomic_int`, the correction is still valid.

```
#include <stdatomic.h>

int compute_sum(atomic_int n)
{
    return n * (n + 1) / 2;
}
```

### Atomic load and store sequence not atomic

#### Issue

**Atomic load and store sequence not atomic** occurs when you use these functions to load, and then store an atomic variable.

- C functions:
  - `atomic_load()`
  - `atomic_load_explicit()`
  - `atomic_store()`
  - `atomic_store_explicit()`
- C++ functions:
  - `std::atomic_load()`
  - `std::atomic_load_explicit()`
  - `std::atomic_store()`
  - `std::atomic_store_explicit()`
  - `std::atomic::load()`
  - `std::atomic::store()`

A thread cannot interrupt an atomic load or an atomic store operation on a variable, but a thread can interrupt a store, and then load sequence.

#### Risk

A thread can modify a variable between the load and store operations, resulting in a data race condition.

#### Fix

To read, modify, and store a variable atomically, use a compound assignment operator such as `+=`, `atomic_compare_exchange()` or `atomic_fetch_*`-family functions.

#### Example - Loading Then Storing an Atomic Variable

In this example, variable `flag` of type `atomic_bool` is referenced twice inside the `toggle_flag()` function. The function loads the variable, negates its value, then stores the new value back to the variable. If two threads call `toggle_flag()`, the second thread can access `flag` between the load and store operations of the first thread. `flag` can end up in an incorrect state.

To see this violation, run Polyspace Bug Finder on this code and spcify `gnu4.9` as the compiler. See `Compiler (-compiler)`.

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void init_flag(void)
{
    atomic_init(&flag, false);
}

void toggle_flag(void)
{
    bool temp_flag = atomic_load(&flag);
    temp_flag = !temp_flag;
    atomic_store(&flag, temp_flag); //Noncompliant
}

bool get_flag(void)
{
    return atomic_load(&flag);
}
```

#### Correction — Use Compound Assignment to Modify Variable

One possible correction is to use a compound assignment operator to toggle the value of `flag`. The C standard defines the operation by using ^= as atomic.

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void toggle_flag(void)
{
    flag ^= 1;
}

bool get_flag(void)
{
    return flag;
}
```

## Check Information

**Group:** Rule 14. Concurrency (CON)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CON40-C

# CERT C: Rule CON41-C

Wrap functions that can fail spuriously in a loop

## Description

### Rule Definition

*Wrap functions that can fail spuriously in a loop.*

### Polyspace Implementation

The rule checker checks for **Function that can spuriously fail not wrapped in loop**.

## Examples

### Function that can spuriously fail not wrapped in loop

#### Issue

**Function that can spuriously fail not wrapped in loop** occurs when the following atomic compare and exchange functions that can fail spuriously are called from outside a loop.

- C atomic functions:
  - `atomic_compare_exchange_weak()`
  - `atomic_compare_exchange_weak_explicit()`
- C++ atomic functions:
  - `std::atomic<T>::compare_exchange_weak(T* expected, T desired)`
  - `std::atomic<T>::compare_exchange_weak_explicit(T* expected, T desired, std::memory_order succ, std::memory_order fail)`
  - `std::atomic_compare_exchange_weak(std::atomic<T>* obj, T* expected, T desired)`
  - `std::atomic_compare_exchange_weak_explicit(volatile std::atomic<T>* obj, T* expected, T desired, std::memory_order succ, std::memory_order fail)`

The functions compare the memory contents of the object representations pointed to by `obj` and `expected`. The comparison can spuriously return false even if the memory contents are equal. This spurious failure makes the functions faster on some platforms.

#### Risk

An atomic compare and exchange function that spuriously fails can cause unexpected results and unexpected control flow.

#### Fix

Wrap atomic compare and exchange functions that can spuriously fail in a loop. The loop checks the failure condition after a possible spurious failure.

### Example - `atomic_compare_exchange_weak()` Not Wrapped in Loop

```
#include <stdatomic.h>

extern void reset_count(void);
atomic_int count = ATOMIC_VAR_INIT(0);

void increment_count(void)
{
    int old_count = atomic_load(&count);
    int new_count;
    new_count = old_count + 1;
    if (!atomic_compare_exchange_weak(&count, &old_count, new_count)) //Noncompliant
        reset_count();

}
```

In this example, `increment_count()` uses `atomic_compare_exchange_weak()` to compare `count` and `old_count`. If the counts are equal, `count` is incremented to `new_count`. If they are not equal, the count is reset. When `atomic_compare_exchange_weak()` fails spuriously, the count is reset unnecessarily.

### Correction — Wrap `atomic_compare_exchange_weak()` in a `while` Loop

One possible correction is to wrap the call to `atomic_compare_exchange_weak()` in a `while` loop. The loop checks the failure condition after a possible spurious failure.

```
#include <stdatomic.h>

extern void reset_count(void);
atomic_int count = ATOMIC_VAR_INIT(0);

void increment_count(void)
{
    int old_count = atomic_load(&count);
    int new_count;
    new_count = old_count + 1;

    do {
        reset_count();

    } while (!atomic_compare_exchange_weak(&count, &old_count, new_count));

}
```

## Check Information

**Group:** Rule 14. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CON41-C

# CERT C: Rule CON43-C

Do not allow data races in multithreaded code

## Description

### Rule Definition

*Do not allow data races in multithreaded code.*

### Polyspace Implementation

The rule checker checks for **Data race**.

### Extend Checker

For the rule checker to detect issues, your code must use one of the concurrency primitives recognized by Polyspace for thread creation and shared variable protection. Otherwise, you must explicitly specify tasks, interrupts, critical sections, and other multitasking options in your project configuration. See “Multitasking”.

You can also extend this checker in the following ways:

- Map your multithreading functions to their known POSIX equivalents. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.
- Detect all data races including ones involving atomic operations.

  Polyspace assumes that certain operations are atomic and excludes them from data race checks. See “Define Atomic Operations in Multitasking Code”. These assumptions might not apply to your environment. To extend the data race checkers to include these operations, use the option `-detect-atomic-data-race`. See “Extend Data Race Checkers to Atomic Operations”.

## Examples

### Data race

#### Issue

Data race occurs when:

- Multiple tasks perform unprotected operations on a shared variable.
- At least one task performs a write operation.
- At least one operation is nonatomic. To detect data race on both atomic and nonatomic operations, use the options `-detect-atomic-data-race`. See “Extend Data Race Checkers to Atomic Operations”.

  See also “Define Atomic Operations in Multitasking Code”.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

#### Risk

Data race can result in unpredictable values of the shared variable because you do not control the order of the operations in different tasks.

Data races between two write operations are more serious than data races between a write and read operation. Two write operations can interfere with each other and result in indeterminate values. To identify write-write conflicts, use the filters on the **Detail** column of the **Results List** pane. For these conflicts, the **Detail** column shows the additional line:

```
Variable value may be altered by write-write concurrent access.
```

See “Filter and Group Results in Polyspace Desktop User Interface” or “Filter and Sort Results in Polyspace Access Web Interface”.

#### Fix

To fix this defect, protect the operations on the shared variable using critical sections, temporal exclusion or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon. For an example, see below.

#### Example - Unprotected Operation on Global Variable from Multiple Tasks

```
int var; //Noncompliant
void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
    var++;
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ | |
| **Tasks** on page 2-121 | task1<br>task2<br>task3 | |
| **Critical section details** on page 2-133 | **Starting routine** | **Ending routine** |
| | begin_critical_section | end_critical_section |

On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2,task3
  -critical-section-begin begin_critical_section:cs1
  -critical-section-end end_critical_section:cs1
```

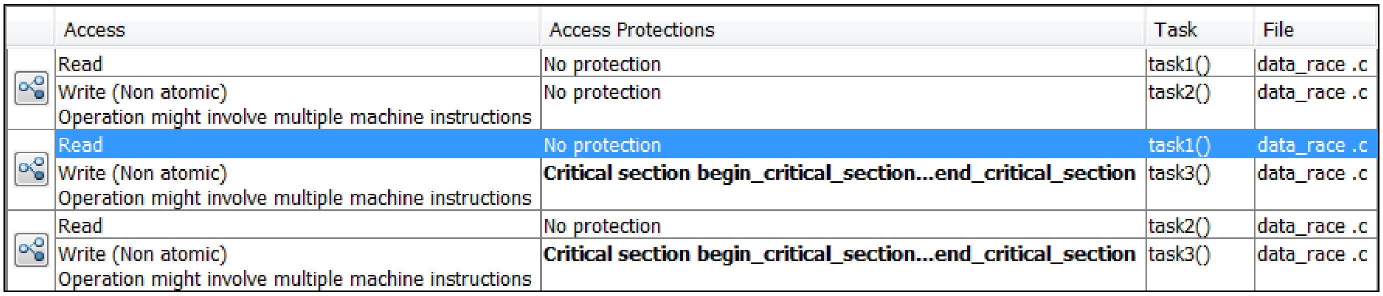
In this example, the tasks `task1`, `task2`, and `task3` call the function `increment`. `increment` contains the operation `var++` that can involve multiple machine instructions including:

- Reading `var`.
- Writing an increased value to `var`.

These machine instructions, when executed from `task1` and `task2`, can occur concurrently in an unpredictable sequence. For example, reading `var` from `task1` can occur either before or after writing to `var` from `task2`. Therefore the value of `var` can be unpredictable.

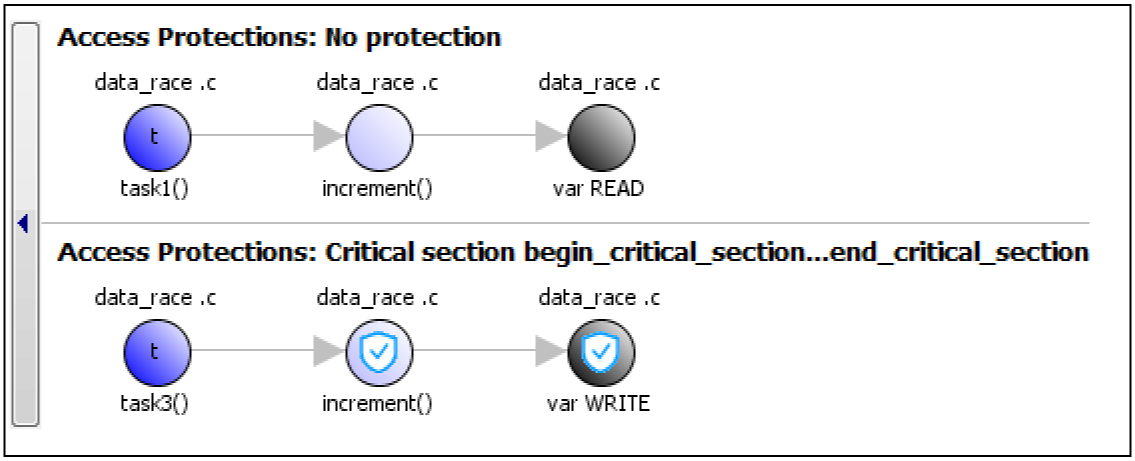
Though `task3` calls `increment` inside a critical section, other tasks do not use the same critical section. The operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

Therefore, the three tasks are operating on a shared variable without common protection. In your result details, you see each pair of conflicting function calls.



| | Access | Access Protections | Task | File |
|---|---|---|---|---|
| | Read | No protection | task1() | data_race .c |
| | Write (Non atomic)<br>Operation might involve multiple machine instructions | No protection | task2() | data_race .c |
| | Read | No protection | task1() | data_race .c |
| | Write (Non atomic)<br>Operation might involve multiple machine instructions | Critical section begin_critical_section...end_critical_section | task3() | data_race .c |
| | Read | No protection | task2() | data_race .c |
| | Write (Non atomic)<br>Operation might involve multiple machine instructions | Critical section begin_critical_section...end_critical_section | task3() | data_race .c |

If you click the icon, you see the function call sequence starting from the entry point to the read or write operation. You also see that the operation starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.



**Correction — Place Operation in Critical Section**

One possible correction is to place the operation in critical section. You can implement the critical section in multiple ways. For instance:

- You can place `var++` in a critical section. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The operation `var++` from the three tasks cannot interfere with each other.

  To implement the critical section, in the function `increment`, place the operation `var++` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      begin_critical_section();
      var++;
      end_critical_section();
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
      increment();
}
```

- You can place the call to `increment` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `increment` from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call `increment` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      var++;
}

void task1(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task2(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

#### Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

| Option | Value |
|---|---|
| **Temporally exclusive tasks** on page 2-136 | `task1 task2 task3` |

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

#### Example - Unprotected Operation in Threads Created with `pthread_create`

```
#include <pthread.h>

pthread_mutex_t count_mutex;
```

```
long long count; //Noncompliant


void* increment_count(void* args)
{
    count = count + 1;
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    c = count;
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

In this example, Bug Finder detects the creation of separate threads with `pthread_create`. The **Data race** defect is raised because the operation `count = count + 1` in the thread with id `thread_increment` conflicts with the operation `c = count` in the thread with id `thread_get`. The variable `count` is accessed in multiple threads without a common protection.

The two conflicting operations are nonatomic. The operation `c = count` is nonatomic on 32-bit targets. See “Define Atomic Operations in Multitasking Code”.

**Correction — Protect Operations with `pthread_mutex_lock` and `pthread_mutex_unlock` Pair**

To prevent concurrent access on the variable `count`, protect operations on `count` with a critical section. Use the functions `pthread_mutex_lock` and `pthread_mutex_unlock` to implement the critical section.

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}
```

```
void* set_count(void *args)
{
    long long c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

## Check Information

**Group:** Rule 14. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CON43-C

# CERT C: Rule MSC30-C

Do not use the rand() function for generating pseudorandom numbers

## Description

### Rule Definition

*Do not use the rand() function for generating pseudorandom numbers.*

### Polyspace Implementation

The rule checker checks for **Use of `rand()` for Generating Pseudorandom Number**.

## Examples

### Use of `rand()` for Generating Pseudorandom Number

**Issue**

This issue occurs when you use the function `rand` for generating pseudorandom numbers.

**Risk**

The function `rand` is cryptographically weak. That is, the numbers generated by `rand` can be predictable. Do not use pseudorandom numbers generated from `rand` for security purposes. When a predictable random value controls the execution flow, your program is vulnerable to attacks.

**Fix**

Use more cryptographically sound pseudorandom number generators (PRNG), such as `CryptGenRandom` (Windows), `OpenSSL/RAND_bytes`(Linux/UNIX), or `random` (POSIX).

**Example - Random Loop Numbers**

```
#include <stdio.h>
#include <stdlib.h>

volatile int rd = 1;
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;

    nloops = rand(); //Noncompliant

    for (j = 0; j < nloops; j++) {
        i = rand(); //Noncompliant
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

This example uses `rand` to generate random numbers `nloops` and `i`. The predictability of these variables makes these function vulnerable to attacks.

#### Correction — Use Stronger PRNG

One possible correction is to replace the vulnerable PRNG with a stronger random number generator. For instance, this code uses the PRNG `random()` from POSIX library. `random` is a much stronger PRNG because it can be seeded by a different number every time it is called.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define TIME_UTC 1
volatile int rd = 1;
int randomWrapper(){
    struct timespec ts;
  if (timespec_get(&ts, TIME_UTC) == 0) {
    /* Handle error */
  }
  srandom(ts.tv_nsec ^ ts.tv_sec);  /* Seed the PRNG */
    return random();
}
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;

    nloops = randomWrapper();

    for (j = 0; j < nloops; j++) {
        i = randomWrapper();
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

## Check Information

**Group:** Rule 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

MSC30-C

# CERT C: Rule MSC32-C

Properly seed pseudorandom number generators

## Description

### Rule Definition

*Properly seed pseudorandom number generators.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Deterministic random output from constant seed**.
- **Predictable random output from predictable seed**.

## Examples

### Deterministic random output from constant seed

#### Issue

**Deterministic random output from constant seed** detects random standard functions that when given a constant seed, have deterministic output.

#### Risk

When some random functions, such as `srand`, `srandom`, and `initstate`, have constant seeds, the results produce the same output every time that your program is run. A hacker can disrupt your program if they know how your program behaves.

#### Fix

Use a different random standard function or use a nonconstant seed.

Some standard random routines are inherently cryptographically weak on page 13-97, and should not be used for security purposes.

#### Example - Random Number Generator Initialization

```
#include <stdlib.h>

void random_num(void)
{
    srand(12345U); //Noncompliant
    /* ... */
}
```

This example initializes a random number generator using `srand` with a constant seed. The random number generation is deterministic, making this function cryptographically weak.

#### Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <stdio.h>

unsigned int random_num_time(void)
{

    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

### Predictable random output from predictable seed

#### Issue

**Predictable random output from predictable seed** looks for random standard functions that use a nonconstant but predictable seed. Examples of predictable seed generators are `time`, `gettimeofday`, and `getpid`.

#### Risk

When you use predictable seed values for random number generation, your random numbers are also predictable. A hacker can disrupt your program if they know how your program behaves.

#### Fix

You can use a different function to generate less predictable seeds.

You can also use a different random number generator that does not require a seed. For example, the Windows API function `rand_s` seeds itself by default. It uses information from the entire system, for example, system time, thread ids, system counter, and memory clusters. This information is more random and a user cannot access this information.

Some standard random routines are inherently cryptographically weak on page 13-97, and should not be used for security purposes.

#### Example - Seed as an Argument

```
#include <stdlib.h>
#include <time.h>

void seed_rng(int seed)
```

```
{
    srand(seed); //Noncompliant
}

int generate_num(void)
{
    seed_rng(time(NULL) + 3);
    /* ... */
}
```

This example uses `srand` to start the random number generator with `seed` as the seed. However, `seed` is predictable because the function `time` generates it. So, an attacker can predict the random numbers generated by `srand`.

**Correction — Use Different Random Number Generator**

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int generate_num(void)
{
    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

## Check Information

**Group:** Rule 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**

"Check for and Review Coding Standard Violations"

### External Websites

MSC32-C

# CERT C: Rule MSC33-C

Do not pass invalid data to the asctime() function

## Description

### Rule Definition

*Do not pass invalid data to the asctime() function.*

### Polyspace Implementation

The rule checker checks for **Use of obsolete standard function**.

## Examples

### Use of obsolete standard function

#### Issue

**Use of obsolete standard function** detects calls to standard function routines that are considered legacy, removed, deprecated, or obsolete by C/C++ coding standards.

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| `asctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |
| `asctime_r` | Deprecated in POSIX.1-2008 | Implementation based on unsafe function `sprintf`. | `strftime` or `asctime_s` |
| `bcmp` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcmp` |
| `bcopy` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcpy` or `memmove` |
| `brk` and `sbrk` | Marked as legacy in SUSv2 and POSIX.1-2001. | | `malloc` |
| `bsd_signal` | Removed in POSIX.1-2008 | | `sigaction` |
| `bzero` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | `memset` |
| `ctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| ctime_r | Deprecated in POSIX.1-2008 | Implementation based on unsafe function sprintf. | strftime or asctime_s |
| cuserid | Removed in POSIX.1-2001. | Not reentrant. Precise functionality not standardized causing portability issues. | getpwuid |
| ecvt and fcvt | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | Not reentrant | snprintf |
| ecvt_r and fcvt_r | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | | snprintf |
| ftime | Removed in POSIX.1-2008 | | time, gettimeofday, clock_gettime |
| gamma, gammaf, gammal | Function not specified in any standard because of historical variations | Portability issues. | tgamma, lgamma |
| gcvt | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | snprintf |
| getcontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| getdtablesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_OPEN_MAX ) |
| gethostbyaddr | Removed in POSIX.1-2008 | Not reentrant | getaddrinfo |
| gethostbyname | Removed in POSIX.1-2008 | Not reentrant | getnameinfo |
| getpagesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_PAGESIZE ) |
| getpass | Removed in POSIX.1-2001. | Not reentrant. | getpwuid |
| getw | Not present in POSIX.1-2001. | | fread |
| getwd | Marked legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | getcwd |
| index | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strchr |
| makecontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| memalign | Appears in SunOS 4.1.3. Not in 4.4 BSD or POSIX.1-2001 | | posix_memalign |
| mktemp | Removed in POSIX.1-2008. | Generated names are predictable and can cause a race condition. | mkstemp removes race risk |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| pthread_attr_getstackaddr and pthread_attr_setstackaddr | | Ambiguities in the specification of the stackaddr attribute cause portability issues | pthread_attr_getstack and pthread_attr_setstack |
| putw | Not present in POSIX.1-2001. | Portability issues. | fwrite |
| qecvt and qfcvt | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | snprintf |
| qecvt_r and qfcvt_r | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | snprintf |
| rand_r | Marked as obsolete in POSIX.1-2008 | | |
| re_comp | BSD API function | Portability issues | regcomp |
| re_exes | BSD API function | Portability issues | regexec |
| rindex | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strrchr |
| scalb | Removed in POSIX.1-2008 | | scalbln, scalblnf, or scalblnl |
| sigblock | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigmask | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigsetmask | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigstack | Interface is obsolete and not implemented on most platforms. | Portability issues. | sigaltstack |
| sigvec | 4.3BSD signal API whose origin is unclear | | sigaction |
| swapcontext | Removed in POSIX.1-2008 | Portability issues. | Use POSIX threads. |
| tmpnam and tmpnam_r | Marked as obsolete in POSIX.1-2008. | This function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined. | mkstemp, tmpfile |
| ttyslot | Removed in POSIX.1-2001. | | |
| ualarm | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | Errors are under-specified | setitimer or POSIX timer_create |
| usleep | Removed in POSIX.1-2008. | | nanosleep |
| utime | SVr4, POSIX.1-2001. POSIX.1-2008 marks as obsolete. | | |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| `valloc` | Marked as obsolete in 4.3BSD.<br>Marked as legacy in SUSv2.<br>Removed from POSIX.1-2001 | | `posix_memalign` |
| `vfork` | Removed from POSIX.1-2008 | Under-specified in previous standards. | `fork` |
| `wcswcs` | This function was not included in the final ISO/IEC 9899:1990/ Amendment 1:1995 (E). | | `wcsstr` |
| `WinExec` | WinAPI provides this function only for 16-bit Windows compatibility. | | `CreateProcess` |
| `LoadModule` | WinAPI provides this function only for 16-bit Windows compatibility. | | `CreateProcess` |

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface".

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Printing Out Time

```
#include <stdio.h>
#include <time.h>

void timecheck_bad(int argc, char *argv[])
{
    time_t ticks;

    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks)); //Noncompliant
}
```

In this example, the function `ctime` formats the current time and prints it out. However, `ctime` was removed after C99 because it does not work on multithreaded programs.

#### Correction — Different Time Function

One possible correction is to use `strftime` instead because this function uses a set buffer size.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

void timecheck_good(int argc, char *argv[])
{
    char outBuff[1025];
    time_t ticks;
    struct tm * timeinfo;

    memset(outBuff, 0, sizeof(outBuff));

    ticks = time(NULL);
    timeinfo = localtime(&ticks);
    strftime(outBuff,sizeof(outBuff),"%I:%M%p.",timeinfo);
    fprintf(stdout, outBuff);
}
```

## Check Information

**Group:** Rule 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C (-cert-c)

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC33-C

# CERT C: Rule MSC37-C

Ensure that control never reaches the end of a non-void function

## Description

### Rule Definition

*Ensure that control never reaches the end of a non-void function.*

### Polyspace Implementation

The rule checker checks for **Missing return statement**.

## Examples

### Missing return statement

#### Issue

**Missing return statement** occurs when a function does not return a value along at least one execution path. If the return type of the function is `void`, this error does not occur.

#### Risk

If a function has a non-`void` return value in its signature, it is expected to return a value. The return value of this function can be used in later computations. If the execution of the function body goes through a path where a `return` statement is missing, the function return value is indeterminate. Computations with this return value can lead to unpredictable results.

#### Fix

In most cases, you can fix this defect by placing the `return` statement at the end of the function body.

Alternatively, you can identify which execution paths through the function body do not have a `return` statement and add a `return` statement on those paths. Often the result details show a sequence of events that indicate this execution path. You can add a `return` statement at an appropriate point in the path. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Missing or invalid return statement error**

```
int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }
 }  //Noncompliant
/* Defect: No return value if n is not 0*/
```

If n is equal to 0, the code does not enter the if statement. Therefore, the function AddSquares does not return a value if n is 0.

**Correction — Place Return Statement on Every Execution Path**

One possible correction is to return a value in every branch of the if...else statement.

```
 int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }

   /*Fix: Place a return statement on branches of if-else */
   else
     return 0;
  }
```

## Check Information

**Group:** Rule 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C (-cert-c)

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

MSC37-C

# CERT C: Rule MSC38-C

Do not treat a predefined identifier as an object if it might only be implemented as a macro

## Description

### Rule Definition

*Do not treat a predefined identifier as an object if it might only be implemented as a macro.*

### Polyspace Implementation

The rule checker checks for **Predefined macro used as an object**.

## Examples

### Predefined macro used as an object

#### Issue

**Predefined macro used as an object** occurs when you use certain identifiers in a way that requires an underlying object to be present. These identifiers are defined as macros. The C Standard does not allow you to redefine them as objects. You use the identifiers in such a way that macro expansion of the identifiers cannot occur.

For instance, you refer to an external variable `errno`:

```
extern int errno;
```

However, `errno` does not occur as a variable but a macro.

The defect applies to these macros: `assert`, `errno`, `math_errhandling`, `setjmp`, `va_arg`, `va_copy`, `va_end`, and `va_start`. The checker looks for the defect only in source files (not header files).

#### Risk

The C11 Standard (Sec. 7.1.4) allows you to redefine most macros as objects. To access the object and not the macro in a source file, you do one of these:

- Redeclare the identifier as an external variable or function.
- For function-like macros, enclose the identifier name in parentheses.

If you try to use these strategies for macros that cannot be redefined as objects, an error occurs.

#### Fix

Do not use the identifiers in such a way that a macro expansion is suppressed.

- Do not redeclare the identifiers as external variables or functions.
- For function-like macros, do not enclose the macro name in parentheses.

#### Example - Use of `assert` as Function

```
#include<assert.h>
typedef void (*err_handler_func)(int);

extern void demo_handle_err(err_handler_func, int);

void func(int err_code) {
    extern void assert(int);    //Noncompliant
    demo_handle_err(&(assert), err_code); //Noncompliant
}
```

In this example, the `assert` macro is redefined as an external function. When passed as an argument to `demo_handle_err`, the identifier `assert` is enclosed in parentheses, which suppresses use of the `assert` macro.

#### Correction — Use `assert` as Macro

One possible correction is to directly use the `assert` macro from `assert.h`. A different implementation of the function `demo_handle_err` directly uses the `assert` macro instead of taking the address of an `assert` function.

```
#include<assert.h>
void demo_handle_err(int err_code) {
    assert(err_code == 0);
}

void func(int err_code) {
    demo_handle_err(err_code);
}
```

## Check Information

**Group:** Rule 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC38-C

# CERT C: Rule MSC39-C

Do not call va_arg() on a va_list that has an indeterminate value

## Description

### Rule Definition

*Do not call va_arg() on a va_list that has an indeterminate value.*

### Polyspace Implementation

The rule checker checks for **Use of indeterminate va_list values**.

## Examples

### Use of indeterminate va_list values

#### Issue

This issue occurs when:

- You use a local `va_list` without initializing it first using `va_start` or `va_copy`.

  You might be using the local `va_list` in `va_arg` or a `vprintf`-like function (function that takes variable number of arguments).
- You use a `va_list` (variable argument list) from a function parameter directly instead of making a copy using `va_copy` and using the copy.

#### Risk

If you use a local `va_list` without initializing it first, the behavior is undefined.

If you pass a `va_list` to another function and use it there, the `va_list` has indeterminate values in the original calling function. Using the `va_list` in the calling function following the function call can produce unexpected results.

#### Fix

Initialize a local `va_list` with `va_start` or `va_copy` before using it.

Pass a `va_list` by reference. In the called function, make a copy of the passed `va_list` and use the copy. You can then continue to access the original `va_list` in the calling function.

#### Example - Direct Use of `va_list` From Another Function

```
#include <stdarg.h>
#include <stdio.h>

int contains_zero(size_t count, va_list ap) {
    for (size_t i = 1; i < count; ++i) {
        if (va_arg(ap, double) == 0.0) {//Noncompliant
            return 1;
        }
    }
    return 0;
}

int print_reciprocals(size_t count, ...) {
    va_list ap;
    va_start(ap, count);

    if (contains_zero(count, ap)) {
        va_end(ap);
        return 1;
    }

    for (size_t i = 0; i < count; ++i) {
        printf("%f ", 1.0 / va_arg(ap, double));
    }

    va_end(ap);
    return 0;
}
```

In this example, the function `print_reciprocals` prints out its variable arguments and uses a helper function `contains_zero` to check if the `va_list` named `ap` contains zero. After `ap` is passed to `contains_zero` by value, the value of `ap` is indeterminate. Attempts to read this indeterminate value in `print_reciprocals` results in unexpected behavior. Polyspace flags the direct of `ap` in th helper function.

#### Correction – Copy `va_list` Obtained from Another Function

To avoid the violation, pass the `va_list` by reference and make a copy of the variable in the `contains_zero` function. Perform further operations on the copy.

```
#include <stdarg.h>
#include <stdio.h>

int contains_zero(size_t count, va_list *ap) {
    va_list ap1;
    va_copy(ap1, *ap);
    for (size_t i = 1; i < count; ++i) {
        if (va_arg(ap1, double) == 0.0) {
            return 1;
        }
    }
    va_end(ap1);
    return 0;
}

int print_reciprocals(size_t count, ...) {
    int status;
    va_list ap;
    va_start(ap, count);

    if (contains_zero(count, &ap)) {
        printf("0 in arguments!\n");
        status = 1;
    } else {
        for (size_t i = 0; i < count; i++) {
            printf("%f ", 1.0 / va_arg(ap, double));
        }
        printf("\n");
        status = 0;
    }

    va_end(ap);
    return status;
}
```

## Check Information

**Group:** Rule 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MSC39-C

# CERT C: Rule MSC40-C

Do not violate constraints

## Description

### Rule Definition

*Do not violate constraints.*

### Polyspace Implementation

The rule checker checks for **Inline constraint not respected**.

## Examples

### Inline constraint not respected

#### Issue

**Inline constraint not respected** occurs when you refer to a file scope modifiable static variable or define a local modifiable static variable in a nonstatic inlined function. The checker considers a variable as modifiable if it is not `const`-qualified.

For instance, `var` is a modifiable `static` variable defined in an `inline` function `func`. `g_step` is a file scope modifiable static variable referred to in the same inlined function.

```
static int g_step;
inline void func (void) {
   static int var = 0;
   var += g_step;
}
```

#### Risk

When you modify a static variable in multiple function calls, you expect to modify the same variable in each call. For instance, each time you call `func`, the same instance of `var1` is incremented but a separate instance of `var2` is incremented.

```
void func(void) {
   static var1 = 0;
   var2 = 0;
   var1++;
   var2++;
}
```

If a function has an inlined and non-inlined definition (in separate files), when you call the function, the C standard allows compilers to use either the inlined or the non-inlined form (see ISO/IEC 9899:2011, sec. 6.7.4). If your compiler uses an inlined definition in one call and the non-inlined definition in another, you are no longer modifying the same variable in both calls. This behavior defies the expectations from a static variable.

#### Fix

Use one of these fixes:

- If you do not intend to modify the variable, declare it as `const`.

  If you do not modify the variable, there is no question of unexpected modification.
- Make the variable non-`static`. Remove the `static` qualifier from the declaration.

  If the variable is defined in the function, it becomes a regular local variable. If defined at file scope, it becomes an extern variable. Make sure that this change in behavior is what you intend.
- Make the function `static`. Add a `static` qualifier to the function definition.

  If you make the function `static`, the file with the inlined definition always uses the inlined definition when the function is called. Other files use another definition of the function. The question of which function definition gets used is not left to the compiler.

#### Example - Static Variable Use in Inlined and External Definition

```
/* file1. c  : contains inline definition of get_random()*/

inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;  //Noncompliant
    static unsigned int m_w = 0xbaddecaf;  //Noncompliant

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}
```

```
/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

In this example, get_random() has an inline definition in file1.c and an external definition in file2.c. When get_random is called in file1.c, compilers are free to choose whether to use the inline or the external definition.

Depending on the definition used, you might or might not modify the version of m_z and m_w in the inlined version of get_random(). This behavior contradicts the usual expectations from a static variable. When you call get_random(), you expect to always modify the same m_z and m_w.

**Correction — Make Inlined Function Static**

One possible correction is to make the inlined get_random() static. Irrespective of your compiler, calls to get_random() in file1.c then use the inlined definition. Calls to get_random() in other files use the external definition. This fix removes the ambiguity about which definition is used and whether the static variables in that definition are modified.

```
/* file1. c  : contains inline definition of get_random()*/

static inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

## Check Information

**Group:** Rule 48. Miscellaneous (MSC)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC40-C

# CERT C: Rule MSC41-C

Never hard code sensitive information

## Description

### Rule Definition

*Never hard code sensitive information.*

### Polyspace Implementation

The rule checker checks for **Hard coded sensitive data**.

## Examples

### Hard coded sensitive data

**Hard coded sensitive data** occurs when data that is potentially sensitive is directly exposed in the code, for instance, as string literals. The checker identifies data as sensitive from their use in certain functions such as password encryption functions.

Following data can be potentially sensitive.

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Host name | • `sethostname`, `setdomainname`, `gethostbyname`, `gethostbyname2`, `getaddrinfo`, `gethostbyname_r`, `gethostbyname2_r` (string argument)<br>• `inet_aton`, `inet_pton`, `inet_net_pton`, `inet_addr`, `inet_network` (string argument)<br>• `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (2nd argument) |
| Password | • `CreateProcessWithLogonW`, `LogonUser` (1st argument)<br>• `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (3rd argument) |

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Database | • MySQL: mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (4th argument)<br>• SQLite: sqlite3_open, sqlite3_open16, sqlite3_open_v2 (1st argument)<br>• PostgreSQL: PQconnectdb<br>• Microsoft SQL: SQLDriverConnect (3rd argument) |
| User name | • getpw, getpwnam, getpwnam_r, getpwuid, getpwuid_r |
| Salt | crypt, crypt_r (2nd argument) |
| Cryptography keys and initialization vectors | OpenSSL:<br>• EVP_CipherInit, EVP_EncryptInit, EVP_DecryptInit (3rd argument)<br>• EVP_CipherInit_ex, EVP_EncryptInit_ex, EVP_DecryptInit_ex (4th argument) |
| Seed | • srand, srandom, initstate (1st argument)<br>• OpenSSL: RAND_seed, RAND_add |

**Risk**

Information that is hardcoded can be queried from binaries generated from the code.

**Fix**

Avoid hard coding sensitive information.

**Example – Sensitive Data Exposed Through String Literals**

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"
const char *host = "localhost";
char *user = "guest";
char *passwd;

DbHandle connect_to_database_server(const char *db)
```

```
{
    passwd = (char*)"guest";
    return (DbHandle)
        mysql_real_connect (sql, host, user, passwd, db, 0, 0x0, 0); //Noncompliant
}
```

In this example, the `mysql_real_connect` arguments `host` (host name), `user` (user name), and `passwd` (password) are string literals and directly exposed in the code.

Querying the generated binary for ASCII strings can reveal this information.

**Correction – Read Sensitive Data from Secured Configuration Files**

One possible correction is to read the data from a configuration file. In the following corrected example, the call to function `connect_to_database_server_init` presumably reads the host name, user name, and password into its arguments from a secured configuration file.

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"

DbHandle connect_to_database_server(const char *db)
{
    const char *host_from_cfg;
    const char *user_from_cfg;
    const char *passwd_from_cfg;
    const char *db_from_cfg;
    if (connect_to_database_server_init(&host_from_cfg,
                                        &user_from_cfg,
                                        &passwd_from_cfg,
                                        &db_from_cfg))
    {
        return (DbHandle)
            mysql_real_connect (sql, host_from_cfg, user_from_cfg,
                          passwd_from_cfg, db_from_cfg,  0, 0x0, 0);
    }
    else
        return (DbHandle)0x0;
}
```

## Check Information

**Group:** Rule 48. Miscellaneous (MSC)

# Version History

**Introduced in R2020a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC41-C

# CERT C: Rule POS30-C

Use the readlink() function properly

## Description

### Rule Definition

*Use the readlink() function properly.*

### Polyspace Implementation

The rule checker checks for **Misuse of readlink()**.

## Examples

### Misuse of readlink()

#### Issue

**Misuse of readlink()** occurs when you pass a buffer size argument to `readlink()` that does not leave space for a null terminator in the buffer.

For instance:

```
size_t len = readlink("/usr/bin/perl", buf, sizeof(buf));
```

The third argument is exactly equal to the size of the second argument. For large enough symbolic links, this use of `readlink()` does not leave space to enter a null terminator.

#### Risk

The `readlink()` function copies the content of a symbolic link (first argument) to a buffer (second argument). However, the function does not append a null terminator to the copied content. After using `readlink()`, you must explicitly add a null terminator to the buffer.

If you fill the entire buffer when using `readlink`, you do not leave space for this null terminator.

#### Fix

When using the `readlink()` function, make sure that the third argument is one less than the buffer size.

Then, append a null terminator to the buffer. To determine where to add the null terminator, check the return value of `readlink()`. If the return value is -1, an error has occurred. Otherwise, the return value is the number of characters (bytes) copied.

#### Example - Incorrect Size Argument of `readlink`

```
#include <unistd.h>

#define SIZE1024 1024

extern void display_path(const char *);
```

```
void func() {
    char buf[SIZE1024];
    ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf)); //Noncompliant
    if (len > 0) {
        buf[len - 1] = '\0';
    }
    display_path(buf);
}
```

In this example, the third argument of `readlink` is exactly the size of the buffer (second argument). If the first argument is long enough, this use of `readlink` does not leave space for the null terminator.

Also, if no characters are copied, the return value of `readlink` is 0. The following statement leads to a buffer underflow when `len` is 0.

```
buf[len - 1] = '\0';
```

#### Correction — Make Sure Size Argument is One Less Than Buffer Size

One possible correction is to make sure that the third argument of `readlink` is one less than size of the second argument.

The following corrected code also accounts for `readlink` returning 0.

```
#include <stdlib.h>
#include <unistd.h>

#define fatal_error() abort()
#define SIZE1024 1024

extern void display_path(const char *);

void func() {
    char buf[SIZE1024];
    ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf) - 1);
    if (len != -1) {
        buf[len] = '\0';
        display_path(buf);
    }
    else {
        /* Handle error */
        fatal_error();
    }
}
```

## Check Information

**Group:** Rule 50. POSIX (POS)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

POS30-C

# CERT C: Rule POS34-C

Do not call putenv() with a pointer to an automatic variable as the argument

## Description

### Rule Definition

*Do not call putenv() with a pointer to an automatic variable as the argument.*

### Polyspace Implementation

The rule checker checks for **Use of automatic variable as putenv-family function argument**.

## Examples

### Use of automatic variable as putenv-family function argument

#### Issue

**Use of automatic variable as putenv-family function argument** occurs when the argument of a putenv-family function is a local variable with automatic duration.

#### Risk

The function putenv(char *string) inserts a pointer to its supplied argument into the environment array, instead of making a copy of the argument. If the argument is an automatic variable, its memory can be overwritten after the function containing the putenv() call returns. A subsequent call to getenv() from another function returns the address of an out-of-scope variable that cannot be dereferenced legally. This out-of-scope variable can cause environment variables to take on unexpected values, cause the program to stop responding, or allow arbitrary code execution vulnerabilities.

#### Fix

Use setenv()/unsetenv() to set and unset environment variables. Alternatively, use putenv-family function arguments with dynamically allocated memory, or, if your application has no reentrancy requirements, arguments with static duration. For example, a single thread execution with no recursion or interrupts does not require reentrancy. It cannot be called (reentered) during its execution.

#### Example - Automatic Variable as Argument of putenv()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    char env[SIZE1024];
    int retval = sprintf(env, "TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
        /* Handle error */
    }
    /* Environment variable TEST is set using putenv().
    The argument passed to putenv is an automatic variable. */
    retval = putenv(env);    //Noncompliant
    if (retval != 0) {
        /* Handle error */
    }
}
```

In this example, `sprintf()` stores the character string `TEST=var` in `env`. The value of the environment variable `TEST` is then set to `var` by using `putenv()`. Because `env` is an automatic variable, the value of `TEST` can change once `func()` returns.

#### Correction — Use `static` Variable for Argument of `putenv()`

Declare `env` as a static-duration variable. The memory location of `env` is not overwritten for the duration of the program, even after `func()` returns.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024
void func(int var)
{
    /* static duration variable */
    static char env[SIZE1024];
    int retval = sprintf(env,"TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
        /* Handle error */
    }

    /* Environment variable TEST is set using putenv() */
    retval=putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
}
```

#### Correction — Use `setenv()` to Set Environment Variable Value

To set the value of `TEST` to `var`, use `setenv()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    /* Environment variable TEST is set using setenv() */
    int retval = setenv("TEST", var ? "1" : "0", 1);

    if (retval != 0) {
        /* Handle error */
    }
}
```

### Check Information

**Group:** Rule 50. POSIX (POS)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

POS34-C

# CERT C: Rule POS35-C

Avoid race conditions while checking for the existence of a symbolic link

## Description

### Rule Definition

*Avoid race conditions while checking for the existence of a symbolic link.*

### Polyspace Implementation

The rule checker checks for **File access between time of check and use (TOCTOU)**.

## Examples

### File access between time of check and use (TOCTOU)

#### Issue

**File access between time of check and use (TOCTOU)** detects race condition issues between checking the existence of a file or folder, and using a file or folder.

#### Risk

An attacker can access and manipulate your file between your check for the file and your use of a file. Symbolic links are particularly risky because an attacker can change where your symbolic link points.

#### Fix

Before using a file, do not check its status. Instead, use the file and check the results afterward.

#### Example - Check File Before Using

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    if (access(log_path, W_OK)==0) {
        FILE* f = fopen(log_path, "w"); //Noncompliant
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

In this example, before opening and using the file, the function checks if the file exists. However, an attacker can change the file between the first and second lines of the function.

#### Correction — Open Then Check

One possible correction is to open the file, and then check the existence and contents afterward.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    int fd = open(log_path, O_WRONLY);
    if (fd!=-1) {
        FILE *f = fdopen(fd, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

### Check Information

**Group:** Rule 50. POSIX (POS)

## Version History

**Introduced in R2019a**

### See Also

Check SEI CERT-C (-cert-c)

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

POS35-C

# CERT C: Rule POS36-C

Observe correct revocation order while relinquishing privileges

## Description

### Rule Definition

*Observe correct revocation order while relinquishing privileges.*

### Polyspace Implementation

The rule checker checks for **Bad order of dropping privileges**.

## Examples

### Bad order of dropping privileges

#### Issue

**Bad order of dropping privileges** checks the order of privilege drops. If you drop higher elevated privileges before dropping lower elevated privileges, Polyspace raises a defect. For example dropping elevated primary group privileges before dropping elevated ancillary group privileges.

#### Risk

If you drop privileges in the wrong order, you can potentially drop higher privileges that you need to drop lower privileges. The incorrect order can mean, privileges are not dropped, compromising the security of your program.

#### Fix

Respect this order of dropping elevated privileges:

- Drop (elevated) ancillary group privileges, then drop (elevated) primary group privileges.
- Drop (elevated) primary group privileges, then drop (elevated) user privileges.

#### Example - Dropping User Privileges First

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()

static void sanitize_privilege_drop_check(uid_t olduid, gid_t oldgid)
{
    if (seteuid(olduid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
    if (setegid(oldgid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
}
void badprivilegedroporder(void) {
    uid_t
        newuid = getuid(),
        olduid = geteuid();
    gid_t
        newgid = getgid(),
        oldgid = getegid();

    if (setuid(newuid) == -1) {
        /* handle error condition */
        fatal_error();
    }
    if (setgid(newgid) == -1)  {  //Noncompliant
        /* handle error condition */
        fatal_error();
    }
    if (olduid == 0) {
        /* drop ancillary groups IDs only possible for root */
        if (setgroups(1, &newgid) == -1) { //Noncompliant
            /* handle error condition */
            fatal_error();
        }
    }

    sanitize_privilege_drop_check(olduid, oldgid);
}
```

In this example, there are two privilege drops made in the incorrect order. `setgid` attempts to drop group privileges. However, `setgid` requires the user privileges, which were dropped previously using `setuid`, to perform this function. After dropping group privileges, this function attempts to drop ancillary groups privileges by using `setgroups`. This task requires the higher primary group privileges that were dropped with `setgid`. At the end of this function, it is possible to regain group privileges because the order of dropping privileges was incorrect.

#### Correction — Reverse Privilege Drop Order

One possible correction is to drop the lowest level privileges first. In this correction, ancillary group privileges are dropped, then primary group privileges are dropped, and finally user privileges are dropped.

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()

static void sanitize_privilege_drop_check(uid_t olduid, gid_t oldgid)
{
    if (seteuid(olduid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
    if (setegid(oldgid) != -1)
    {
        /* Privileges can be restored, handle error */
        fatal_error();
    }
}
void badprivilegedroporder(void) {
    uid_t
        newuid = getuid(),
        olduid = geteuid();
    gid_t
        newgid = getgid(),
        oldgid = getegid();

    if (olduid == 0) {
        /* drop ancillary groups IDs only possible for root */
        if (setgroups(1, &newgid) == -1) {
            /* handle error condition */
            fatal_error();
        }
    }
    if (setgid(getgid()) == -1)  {
        /* handle error condition */
        fatal_error();
    }
    if (setuid(getuid()) == -1) {
        /* handle error condition */
        fatal_error();
    }

    sanitize_privilege_drop_check(olduid, oldgid);
}
```

## Check Information

**Group:** Rule 50. POSIX (POS)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

POS36-C

# CERT C: Rule POS37-C

Ensure that privilege relinquishment is successful

## Description

### Rule Definition

*Ensure that privilege relinquishment is successful.*

### Polyspace Implementation

The rule checker checks for **Privilege drop not verified**.

## Examples

### Privilege drop not verified

**Issue**

**Privilege drop not verified** detects calls to functions that relinquish privileges. If you do not verify that the privileges were dropped before the end of your function, a defect is raised.

**Risk**

If privilege relinquishment fails, an attacker can regain elevated privileges and have more access to your program than intended. This security hole can cause unexpected behavior in your code if left open.

**Fix**

Before the end of scope, verify that the privileges that you dropped were actually dropped.

**Example - Drop Privileges Within a Function**

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()
extern int need_more_privileges;

void missingprivilegedropcheck()
{
    /* Code intended to run with elevated privileges */

    /* Temporarily drop elevated privileges */
    if (seteuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges */
    if (need_more_privileges) {
        /* Restore elevated privileges */
        if (seteuid(0) != 0) {
            /* Handle error */
            fatal_error();
        }
        /* Code intended to run with elevated privileges */
    }

    /* ... */

    /* Permanently drop elevated privileges */
    if (setuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges */
} //Noncompliant
```

In this example, privileges are elevated and dropped to run code with the intended privilege level. When privileges are dropped, the privilege level before exiting the function body is not verified. A malicious attacker can regain their elevated privileges.

**Correction — Verify Privilege Drop**

One possible correction is to use `setuid` to verify that the privileges were dropped.

```
#define _BSD_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>
#define fatal_error() abort()
extern int need_more_privileges;

void missingprivilegedropcheck()
{
    /* Store the privileged ID for later verification */
    uid_t privid = geteuid();

    /* Code intended to run with elevated privileges   */

    /* Temporarily drop elevated privileges */
    if (seteuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges  */

    if (need_more_privileges) {
        /* Restore elevated Privileges */
        if (seteuid(privid) != 0) {
            /* Handle error */
            fatal_error();
        }
        /* Code intended to run with elevated privileges   */
    }

    /* ... */

    /* Restore privileges if needed */
    if (geteuid() != privid) {
        if (seteuid(privid) != 0) {
            /* Handle error */
            fatal_error();
        }
    }

    /* Permanently drop privileges */
    if (setuid(getuid()) != 0) {
        /* Handle error */
        fatal_error();
    }

    if (setuid(0) != -1) {
        /* Privileges can be restored, handle error */
        fatal_error();
    }

    /* Code intended to run with lower privileges; */
}
```

### Check Information

**Group:** Rule 50. POSIX (POS)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

POS37-C

# CERT C: Rule POS38-C

Beware of race conditions when using fork and file descriptors

## Description

### Rule Definition

*Beware of race conditions when using fork and file descriptors.*

### Polyspace Implementation

The rule checker checks for **File descriptor exposure to child process**.

## Examples

### File descriptor exposure to child process

#### Issue

**File descriptor exposure to child process** occurs when a process is forked and the child process uses file descriptors inherited from the parent process.

#### Risk

When you fork a child process, file descriptors are copied from the parent process, which means that you can have concurrent operations on the same file. Use of the same file descriptor in the parent and child processes can lead to race conditions that may not be caught during standard debugging. If you do not properly manage the file descriptor permissions and privileges, the file content is vulnerable to attacks targeting the child process.

#### Fix

Check that the file has not been modified before forking the process. Close all inherited file descriptors and reopen them with stricter permissions and privileges, such as read-only permission.

#### Example - File Descriptor Accessed from Forked Process

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <fcntl.h>
# include <sys/types.h>
# include <sys/stat.h>



const char *test_file="/home/user/test.txt";

void func(void)
{
    char c;
    pid_t pid;
    /* create file descriptor in read and write mode */
    int fd = open(test_file, O_RDWR);
    if (fd == -1)
    {
        /* Handle error */
        abort();
    }
    /* fork process */
    pid = fork();
    if (pid == -1)
    {
        /* Handle error */
        abort();
    }
    else if (pid == 0)
    {   /* Child process accesses file descriptor inherited
        from parent process */
        (void)read(fd, &c, 1); //Noncompliant
    }
    else
    {   /* Parent process access same file descriptor as
        child process */
        (void)read(fd, &c, 1);
    }
}
```

In this example, a file descriptor `fd` is created in read and write mode. The process is then forked. The child process inherits and accesses `fd` with the same permissions as the parent process. A race condition exists between the parent and child processes. The contents of the file is vulnerable to attacks through the child process.

**Correction — Close and Reopen Inherited File Descriptor**

After you create the file descriptor, check the file for tampering. Then, close the inherited file descriptor in the child process and reopen it in read-only mode.

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <fcntl.h>
# include <sys/types.h>
# include <sys/stat.h>


const char *test_file="/home/user/test.txt";

void func(void)
{
    char c;
    pid_t pid;

    /* Get the state of file for further file tampering checking */

    /* create file descriptor in read and write mode */
    int fd = open(test_file, O_RDWR);
    if (fd == -1)
    {
        /* Handle error */
        abort();
    }

    /* Be sure the file was not tampered with while opening */

    /* fork process */

    pid = fork();
    if (pid == -1)
    {
        /* Handle error */
        (void)close(fd);
        abort();
    }
    else if (pid == 0)
    {  /* Close file descriptor in child process and repoen
        it in read only mode */

        (void)close(fd);
        fd = open(test_file, O_RDONLY);
        if (fd == -1)
        {
            /* Handle error */
            abort();
        }


        (void)read(fd, &c, 1);
        (void)close(fd);
    }
    else
    {  /* Parent acceses original file descriptor */
        (void)read(fd, &c, 1);
        (void)close(fd);
    }
}
```

## Check Information

**Group:** Rule 50. POSIX (POS)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

POS38-C

# CERT C: Rule POS39-C

Use the correct byte ordering when transferring data between systems

## Description

### Rule Definition

*Use the correct byte ordering when transferring data between systems.*

### Polyspace Implementation

The rule checker checks for **Missing byte reordering when transferring data**.

## Examples

### Missing byte reordering when transferring data

#### Issue

**Missing byte reordering when transferring data** occurs when you do not use a byte ordering function:

- Before sending data to a network socket.
- After receiving data from a network socket.

#### Risk

Some system architectures implement little endian byte ordering (least significant byte first), and other systems implement big endian (most significant byte first). If the endianness of the sent data does not match the endianness of the receiving system, the value returned when reading the data is incorrect.

#### Fix

After receiving data from a socket, use a byte ordering function such as `ntohl()`. Before sending data to a socket, use a byte ordering function such as `htonl()` .

#### Example - Data Transferred Without Byte Reordering

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <byteswap.h>
#include <unistd.h>
#include <string.h>


unsigned int func(int sock, int server)
{
    unsigned int num;   /* assume int is 32-bits */
    if (server)
    {
        /* Server side */
        num = 0x17;
        /* Endianness of server host may not match endianness of network. */
        if (send(sock, (void *)&num, sizeof(num), 0) < (int)sizeof(num))  //Noncompliant
        {
            /* Handle error */
        }
        return 0;
    }
    else {
        /* Endianness of client host may not match endianness of network. */
        if (recv (sock, (void *)&num, sizeof(num), 0) < (int) sizeof(num))
        {
            /* Handle error */
        }

        /* Comparison may be inaccurate */
        if (num> 255)   //Noncompliant
        {
            return 255;
        }
        else
        {
            return num;
        }
    }
}
```

In this example, variable num is assigned hexadecimal value 0x17 and is sent over a network to the client from the server. If the server host is little endian and the network is big endian, num is transferred as 0x17000000. The client then reads an incorrect value for num and compares it to a local numeric value.

**Correction — Use Byte Ordering Function**

Before sending num from the server host, use htonl() to convert from host to network byte ordering. Similarly, before reading num on the client host, use ntohl() to convert from network to host byte ordering.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <byteswap.h>
#include <unistd.h>
#include <string.h>

unsigned int func(int sock, int server)
{
    unsigned int num;   /* assume int is 32-bits */
    if (server)
    {
        /* Server side */
        num = 0x17;

        /* Convert to network byte order. */
        num = htonl(num);
        if (send(sock, (void *)&num, sizeof(num), 0) < (int)sizeof(num))
        {
            /* Handle error */
        }
        return 0;
    }
    else {
        if (recv (sock, (void *)&num, sizeof(num), 0) < (int) sizeof(num))
        {
            /* Handle error */
        }

        /* Convert to host byte order. */
        num = ntohl(num);
        if (num > 255)
        {
            return 255;
        }
        else
        {
            return num;
        }
    }
}
```

## Check Information

**Group:** Rule 50. POSIX (POS)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

POS39-C

# CERT C: Rule POS44-C

Do not use signals to terminate threads

## Description

### Rule Definition

*Do not use signals to terminate threads.*

### Polyspace Implementation

The rule checker checks for **Use of signal to kill thread**.

## Examples

### Use of signal to kill thread

#### Issue

**Use of signal to kill thread** occurs when you use an uncaught signal to kill a thread. For instance, you use the POSIX function `pthread_kill` and send the signal `SIGTERM` to kill a thread.

#### Risk

Sending a signal kills the entire process instead of just the thread that you intend to kill.

For instance, the `pthread_kill` specifications state that if the disposition of a signal is to terminate, this action affects the entire process.

#### Fix

Use other mechanisms that are intended to kill specific threads.

For instance, use the POSIX function `pthread_cancel` to terminate a specific thread.

#### Example - Use of `pthread_kill` to Terminate Threads

```
#include <signal.h>
#include <pthread.h>

void* func(void *foo) {
  /* Execution of thread */
}

int main(void) {
  int result;
  pthread_t thread;

  if ((result = pthread_create(&thread, NULL, func, 0)) != 0) {
  }
  if ((result = pthread_kill(thread, SIGTERM)) != 0) { //Noncompliant
  }

  /* This point is not reached because the process terminates in pthread_kill() */
  return 0;
}
```

In this example, the `pthread_kill` function sends the signal `SIGTERM` to kill a thread. The signal kills the entire process instead of the thread previously created with `pthread_create`.

#### Correction — Use `pthread_cancel` to Terminate Threads

One possible correction is to use the `pthread_cancel` function. The `pthread_cancel` terminates a thread specified by its first argument at a specific cancellation point or immediately, depending on the thread's cancellation type.

```
#include <signal.h>
#include <pthread.h>

void* func(void *foo) {
  /* Execution of thread */
}

int main(void) {
  int result;
  pthread_t thread;

  if ((result = pthread_create(&thread, NULL, func, 0)) != 0) {
    /* Handle Error */
  }
  if ((result = pthread_cancel(thread)) != 0) {
    /* Handle Error */
  }

  /* Continue executing */

  return 0;
}
```

See also:

- `pthread_cancel` for more information on cancellation types.
- Pthreads for functions that are allowed to be cancellation points.

## Check Information

**Group:** Rule 50. POSIX (POS)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

POS44-C

# CERT C: Rule POS47-C

Do not use threads that can be canceled asynchronously

## Description

### Rule Definition

*Do not use threads that can be canceled asynchronously.*

### Polyspace Implementation

The rule checker checks for **Asynchronously cancellable thread**.

## Examples

### Asynchronously cancellable thread

#### Issue

This issue occurs when you use `pthread_setcanceltype` with argument `PTHREAD_CANCEL_ASYNCHRONOUS` to set the cancellability type of a calling thread to asynchronous (or immediate) . An asynchronously cancellable thread can be cancelled at any time, usually immediately upon receiving a cancellation request.

#### Risk

The calling thread might be cancelled in an unsafe state that could result in a resources leak, a deadlock, a data race, data corruption, or unpredictable behavior.

#### Fix

Remove the call to `pthread_setcanceltype` with argument `PTHREAD_CANCEL_ASYNCHRONOUS` to use the default cancellability type `PTHREAD_CANCEL_DEFERRED` instead. With the default cancellability type, the thread defers cancellation requests until it calls a function that is a cancellation point.

#### Example - Cancellability Type of Thread Set to Asynchronous

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static int fatal_error(void)
{
    exit(1);
}


volatile int a = 5;
volatile int b = 10;

pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;

void* swap_values_thread(void* dummy)
{
    int i;
    int c;
    int result;
    if ((result =
             pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &i)) != 0) { //Noncompliant
```

```
        /* handle error */
        fatal_error();
    }
    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
        c = b;
        b = a;
        a = c;
        if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
    }
    return NULL;
}

int main(void)
{
    int result;
    pthread_t worker;

    if ((result = pthread_create(&worker, NULL, swap_values_thread, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }

    /* Additional code */

    if ((result = pthread_cancel(worker)) != 0) {
        /* handle error */
        fatal_error();
    }


    if ((result = pthread_join(worker, 0)) != 0) {
        /* handle error */
        fatal_error();
    }

    if ((result = pthread_mutex_lock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }
    printf("a: %i | b: %i", a, b);
    if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }

    return 0;
}
```

In this example, the cancellability type of the `worker` thread is set to asynchronous. The mutex `global_lock` helps ensure that the `worker` and `main` threads do not access variables `a` and `b` at the same time. However, the `worker` thread might be cancelled while holding `global_lock`, and the `main` thread will never acquire `global_lock`, which results in a deadlock.

#### Correction — Use the Default Cancellability Type

One possible correction is to remove the call to `pthread_setcanceltype`. By default, the cancellability type of a new thread is set to `PTHREAD_CANCEL_DEFERRED`. The `worker` thread defers cancellation requests until it calls a function that is a cancellation point.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static int fatal_error(void)
{
    exit(1);
}


volatile int a = 5;
volatile int b = 10;

pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;

void* swap_values_thread(void* dummy)
{
    int i;
    int c;
    int result;
    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
        c = b;
        b = a;
        a = c;
        if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
            /* handle error */
            fatal_error();
        }
    }
    return NULL;
}

int main(void)
{
    int result;
    pthread_t worker;

    if ((result = pthread_create(&worker, NULL, swap_values_thread, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }

    /* Additional code */

    if ((result = pthread_cancel(worker)) != 0) {
        /* handle error */
        fatal_error();
    }


    if ((result = pthread_join(worker, 0)) != 0) {
        /* handle error */
        fatal_error();
    }

    if ((result = pthread_mutex_lock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }
    printf("a: %i | b: %i", a, b);
    if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
        /* handle error */
        fatal_error();
    }

    return 0;
}
```

## Check Information

**Group:** Rule 50. POSIX (POS)

# Version History

**Introduced in R2020a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**
POS47-C

# CERT C: Rule POS48-C

Do not unlock or destroy another POSIX thread's mutex

## Description

### Rule Definition

*Do not unlock or destroy another POSIX thread's mutex.*

### Polyspace Implementation

The rule checker checks for **Destruction of locked mutex**.

## Examples

### Destruction of locked mutex

#### Issue

**Destruction of locked mutex** occurs when a task destroys a mutex after it is locked (and before it is unlocked). The locking and destruction can happen in the same task or different tasks.

#### Risk

A mutex is locked to protect shared variables from concurrent access. If a mutex is destroyed in the locked state, the protection does not apply.

#### Fix

To fix this defect, destroy the mutex only after you unlock it. It is a good design practice to:

- Initialize a mutex *before* creating the threads where you use the mutex.
- Destroy a mutex *after* joining the threads that you created.

On the **Result Details** pane, you see two events, the locking and destruction of the mutex, and the tasks that initiated the events. To navigate to the corresponding line in your source code, click the event.

#### Example - Locking and Destruction in Different Tasks

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
  pthread_mutex_unlock (&lock3);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_destroy (&lock3); //Noncompliant
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

In this example, after task `t0` locks the mutex `lock3`, task `t1` can destroy it. The destruction occurs if the following events happen in sequence:

1. `t0` acquires `lock3`.
2. `t0` releases `lock2`.
3. `t0` releases `lock1`.
4. `t1` acquires the lock `lock1` released by `t0`.
5. `t1` acquires the lock `lock2` released by `t0`.
6. `t1` destroys `lock3`.

For simplicity, this example uses a mix of automatic and manual concurrency detection. The tasks `t0` and `t1` are manually specified as entry points by using the option `Tasks (-entry-points)`. The critical sections are implemented through primitives `pthread_mutex_lock` and `pthread_mutex_unlock` that the software detects automatically. In practice, for entry point specification (thread creation), you will use primitives such as `pthread_create`. The next example shows how the defect can appear when you use `pthread_create`.

#### Correction — Place Lock-Unlock Pair Together in Same Critical Section as Destruction

The locking and destruction of `lock3` occurs inside the critical section imposed by `lock1` and `lock2`, but the unlocking occurs outside. One possible correction is to place the lock-unlock pair in the same critical section as the destruction of the mutex. Use one of these critical sections:

- Critical section imposed by `lock1` alone.
- Critical section imposed by `lock1` and `lock2`.

In this corrected code, the lock-unlock pair and the destruction is placed in the critical section imposed by `lock1` and `lock2`. When `t0` acquires `lock1` and `lock2`, `t1` has to wait for their release before it executes the instruction `pthread_mutex_destroy (&lock3);`. Therefore, `t1` cannot destroy mutex `lock3` in the locked state.

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);

  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);

  pthread_mutex_destroy (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

**Example - Locking and Destruction in Start Routine of Thread**

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_exit((void*) 0);
}

void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_destroy(&lock); //Noncompliant
    pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Thread that initializes mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use mutex for atomic operation*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   pthread_exit(NULL);
}
```

In this example, four threads are created. The threads are assigned different actions.

- The first thread `callThd[0]` initializes the mutex `lock`.
- The second and third threads, `callThd[1]` and `callThd[2]`, perform an atomic operation protected by the mutex `lock`.
- The fourth thread `callThd[3]` destroys the mutex `lock`.

The threads can interrupt each other. Therefore, immediately after the second or third thread locks the mutex, the fourth thread can destroy it.

### Correction — Initialize and Destroy Mutex Outside Start Routine

One possible correction is to initialize and destroy the mutex in the `main` function outside the start routine of the threads. The threads perform only the atomic operation. You need two fewer threads because the mutex initialization and destruction threads are not required.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 2
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_work(void *arg) {
   pthread_mutex_lock (&lock);
   atomic_operation();
   pthread_mutex_unlock (&lock);
   pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize mutex */
   pthread_mutex_init(&lock, NULL);

   for(i=0; i<NUMTHREADS; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy mutex */
   pthread_mutex_destroy(&lock);

   pthread_exit(NULL);
}
```

#### Correction — Use A Second Mutex To Protect Lock-Unlock Pair and Destruction

Another possible correction is to use a second mutex and protect the lock-unlock pair from the destruction. This corrected code uses the mutex `lock2` to achieve this protection. The second mutex is initialized in the `main` function outside the start routine of the threads.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
pthread_mutex_t lock2;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}
```

```
void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_destroy(&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}


int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize second mutex */
   pthread_mutex_init(&lock2, NULL);

   /* Thread that initializes first mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use first mutex for atomic operation */
   /* The threads use second mutex to protect first from destruction in locked state*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys first mutex */
   /* The thread uses the second mutex to prevent destruction of locked mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);


   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy second mutex */
   pthread_mutex_destroy(&lock2);

   pthread_exit(NULL);
}
```

## Check Information

**Group:** Rule 50. POSIX (POS)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

POS48-C

# CERT C: Rule POS49-C

When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed

## Description

### Rule Definition

*When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed.*

### Polyspace Implementation

The rule checker checks for **Data race on adjacent bit fields**.

## Examples

### Data race on adjacent bit fields

#### Issue

Data race on adjacent bit fields occurs when:

- Multiple tasks perform unprotected operations on bit fields that are part of the same structure.

  For instance, a task operates on field `errorFlag1` and another task on field `errorFlag2` in a variable of this type:

  ```
  struct errorFlags {
     unsigned int errorFlag1 : 1;
     unsigned int errorFlag2 : 1;
     ...
  }
  ```

  Suppose that the operations are not atomic with respect to each other. In other words, you have not implemented protection mechanisms to ensure that one operation completes before another begins.
- At least one of the unprotected operations is a write operation.

#### Risk

Adjacent bit fields that are part of the same structure might be stored in one byte in the same memory location. Read or write operations on all variables including bit fields happen one byte or word at a time. To modify only specific bits in a byte, steps similar to this happen in sequence:

1. The byte is loaded into RAM.
2. A mask is created so that only specific bits would be modified to the intended value and the remaining bits remain unchanged.
3. A bitwise OR operation is performed between the copy of the byte in RAM and the mask.
4. The byte with specific bits modified is copied back from RAM.

If two different bit fields are accessed, these four steps have to be performed for each bit field. If the accesses are not protected, all four steps for one bit field might not complete before the four steps for the other begin. As a result, the modification of one bit field might undo the modification of an adjacent bit field. For instance, the modification of `errorFlag1` and `errorFlag2` can happen in the following sequence. Steps marked 1 relate to modification of `errorFlag1` and steps marked 2 relate to that of `errorFlag2`.

1a. The byte with both `errorFlag1` and `errorFlag2` unmodified is copied into RAM, for purposes of modifying `errorFlag1`.

1b. A mask that modifies only `errorFlag1` is bitwise OR-ed with this copy.

2a. The byte containing both `errorFlag1` and `errorFlag2` unmodified is copied into RAM a second time, for purposes of modifying `errorFlag2`.

2b. A mask that modifies only `errorFlag2` is bitwise OR-ed with this second copy.

1c. The version with `errorFlag1` modified is copied back. This version has `errorFlag2` unmodified.

2c The version with `errorFlag2` modified is copied back. This version has `errorFlag1` unmodified and overwrites the previous modification.

#### Fix

To fix this defect, protect the operations on bit fields that are part of the same structure using critical sections, temporal exclusion or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon. For an example, see below.

#### Example - Unprotected Operation on Global Variable from Multiple POSIX Threads

```
#include <stdlib.h>
#include <pthread.h>
#define thread_success 0

typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12; //Noncompliant

void* task1 (void* arg) {
    InterruptConfigbitsProc12.IOFlag = 0;
    //Additional code
}

void* task2 (void* arg) {
    InterruptConfigbitsProc12.SetupFlag = 0;
    //Additional code
}

void main() {
    pthread_t thread1, thread2;
    if(thread_success != pthread_create(&thread1, NULL, task1, NULL)){
        //Handle error
    }
    if(thread_success != pthread_create(&thread2, NULL, task2, NULL)){
        //Handle error
    }
}
```

In this example, the threads with id `thread1` and `thread2` access different bit fields `IOFlag` and `SetupFlag`, which belong to the same structured variable `InterruptConfigbitsProc12`.

#### Correction - Use Critical Sections

One possible correction is to wrap the bit field accesses in a critical section. A critical section lies between a call to a lock function and an unlock function. In this correction, the critical section lies between the calls to functions `pthread_mutex_lock` and `pthread_mutex_unlock`.

```
#include <stdlib.h>
#include <pthread.h>
#define thread_success 0
#define lock_success 0

pthread_mutex_t lock;

typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void* task1 (void* arg) {
    if( lock_success != pthread_mutex_lock(&lock)) {
        //Handle error
    }
    InterruptConfigbitsProc12.IOFlag = 0;
    if( lock_success != pthread_mutex_unlock(&lock)) {
        //Handle error
    }
    //Additional code
}
```

```
void* task2 (void* arg) {
    if( lock_success != pthread_mutex_lock(&lock)) {
        //Handle error
    }
    InterruptConfigbitsProc12.SetupFlag = 0;
    if( lock_success != pthread_mutex_unlock(&lock)) {
        //Handle error
    }
    //Additional code
}

void main() {
    pthread_t thread1, thread2;
    if(thread_success != pthread_create(&thread1, NULL, task1, NULL)){
        //Handle error
    }
    if(thread_success != pthread_create(&thread2, NULL, task2, NULL)){
        //Handle error
    }
}
```

**Correction – Insert Bit Field of Size 0**

You can enter a non bit field member or an unnamed bit field member of size 0 in between two adjacent bit fields that might be accessed concurrently. A non bit field member or size 0 bit field member ensures that the subsequent bit field starts from a new memory location. In this corrected example, the size 0 bit field member ensures that `IOFlag` and `SetupFlag` are stored in distinct memory locations.

```
#include <stdlib.h>
#include <pthread.h>
#define thread_success 0

typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int : 0;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void* task1 (void* arg) {
    InterruptConfigbitsProc12.IOFlag = 0;
    //Additional code
}

void* task2 (void* arg) {
    InterruptConfigbitsProc12.SetupFlag = 0;
    //Additional code
}
```

```
void main() {
    pthread_t thread1, thread2;
    if(thread_success != pthread_create(&thread1, NULL, task1, NULL)){
        //Handle error
    }
    if(thread_success != pthread_create(&thread2, NULL, task2, NULL)){
        //Handle error
    }
}
```

### Check Information

**Group:** Rule 50. POSIX (POS)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

POS49-C

# CERT C: Rule POS50-C

Declare objects shared between POSIX threads with appropriate storage durations

## Description

### Rule Definition

*Declare objects shared between POSIX threads with appropriate storage durations.*

## Examples

### Automatic or thread local variable escaping from a POSIX thread

#### Issue

**Automatic or thread local variable escaping from a POSIX thread** occurs when an automatic or thread local variable is passed by address from one POSIX thread to another without ensuring that the variable stays alive through the duration of the latter thread.

#### Risk

An automatic or thread local variable is allocated on the stack at the beginning of a thread and its lifetime extends till the end of the thread. The variable is not guaranteed to be alive when a different thread accesses it.

For instance, consider the start function of a POSIX thread with these lines:

```
int start_thread(pthread_t *tid) {
   int aVar = 0;
   if(thrd_success != pthread_create(tid, NULL, start_thread_child, &aVar) {
     //...
   }
}
```

The `pthread_create` function creates a child thread with start function `start_thread_child` and passes the address of the automatic variable `aVar`to this function. When this child thread accesses `aVar`, the parent thread might have completed execution and `aVar` is no longer on the stack. The access might result in reading unpredictable values.

#### Fix

When you pass a variable from one thread to another, make sure that the variable lifetime matches or exceeds the lifetime of both threads. You can achieve this synchronization in one of these ways:

- Declare the variable `static` so that it does not go out of stack when the current thread completes execution.
- Dynamically allocate the storage for the variable so that it is allocated on the heap instead of the stack and must be explicitly deallocated. Make sure that the deallocation happens after both threads complete execution.

These solutions require you to create a variable in nonlocal memory. Instead, you can use other solutions such as the `shared` keyword with OpenMP's threading interface that allows you to safely share local variables across threads.

### Example – Local Variable Escaping Thread

```
#include <pthread.h>
#include <stdio.h>

void* create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return NULL;
}

void create_parent_thread(pthread_t *tid) {
  int parentVal = 1;
  int thrd_success;
  if ((thrd_success = pthread_create(tid, NULL, create_child_thread, &parentVal)) != 0) { //Nonco
    /* Handle error */
  }
}

int main(void) {
  pthread_t tid;
  int thrd_success;
  create_parent_thread(&tid);

  if ((thrd_success = thrd_join(tid, NULL)) != 0) {
    /* Handle error */
  }
  return 0;
}
```

In this example, the value `parentVal` is local to the parent thread that starts in `main` and continues into the function `create_parent_thread`. However, in the body of `create_parent_thread`, the address of this local variable is passed to a child thread (the thread with start routine `create_child_thread`). The parent thread might have completed execution and the variable `parentVal` might have gone out of scope when the child thread accesses this variable.

### Correction – Use Static Variables

One possible correction is to declare the variable `parentVal` as `static` so that the variable is on the stack for the entire duration of the program.

```
#include <pthread.h>
#include <stdio.h>

void* create_child_thread(void *childVal) {
  int *res = (int *)childVal;
  printf("Result: %d\n", *res);
  return NULL;
}

void create_parent_thread(pthread_t *tid) {
  static int parentVal = 1;
  int thrd_success;
  if ((thrd_success = pthread_create(tid, NULL, create_child_thread, &parentVal)) != 0) {
    /* Handle error */
  }
}
```

```
int main(void) {
  pthread_t tid;
  int thrd_success;
  create_parent_thread(&tid);

  if ((thrd_success = thrd_join(tid, NULL)) != 0) {
    /* Handle error */
  }
  return 0;
}
```

### Correction – Use Dynamic Memory Allocation

One possible correction is to dynamically allocate storage for variables to be shared across threads and explicitly free the storage after the variable is no longer required.

```
#include <pthread.h>
#include <stlib.h>

void* create_child_thread(void *val) {
  int *res = (int *)val;
  printf("Result: %d\n", *res);
  free(res);
  return NULL;
}

void create_parent_thread(pthread_t *tid) {
  int *val;
  int thrd_success;

  val = malloc(sizeof(int));

  if(!val) {
      *val = 1;
      if ((thrd_success = pthread_create(tid, NULL, create_child_thread, val)) != 0) {
        /* Handle error */
      }
  }
}

int main(void) {
  pthread_t tid;
  int thrd_success;
  create_parent_thread(&tid);

  if ((thrd_success = thrd_join(tid, NULL)) != 0) {
    /* Handle error */
  }
  return 0;
}
```

## Check Information

**Group:** Rule 50. POSIX (POS)

# Version History

**Introduced in R2020a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

POS50-C

# CERT C: Rule POS51-C

Avoid deadlock with POSIX threads by locking in predefined order

## Description

### Rule Definition

*Avoid deadlock with POSIX threads by locking in predefined order.*

### Polyspace Implementation

The rule checker checks for **Deadlock**.

### Extend Checker

You might be using multithreading functions that are not supported by Polyspace. Extend this checker by mapping the functions of your multithreading functions to their known POSIX equivalent. See "Extend Concurrency Defect Checkers to Unsupported Multithreading Environments".

## Examples

### Deadlock

#### Issue

**Deadlock** occurs when multiple tasks are stuck in their critical sections (CS) because:

- Each CS waits for another CS to end.
- The critical sections (CS) form a closed cycle. For example:
  - CS #1 waits for CS #2 to end, and CS #2 waits for CS #1 to end.
  - CS #1 waits for CS #2 to end, CS #2 waits for CS #3 to end and CS #3 waits for CS #1 to end.

Polyspace expects critical sections of code to follow a specific format. A critical section lies between a call to a lock function and a call to an unlock function. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function. Both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

#### Risk

Each task waits for a critical section in another task to end and is unable to proceed. The program can freeze indefinitely.

#### Fix

The fix depends on the root cause of the defect. You can try to break the cyclic order between the tasks in one of these ways:

- Write down all critical sections involved in the deadlock in a certain sequence. Whenever you call the lock functions of the critical sections within a task, respect the order in that sequence. See an example below.
- If one of the critical sections involved in a deadlock occurs in an interrupt, try to disable all interrupts during critical sections in all tasks. See `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Reviewing this defect is an opportunity to check if all operations in your critical section are really meant to be executed as an atomic block. It is a good practice to keep critical sections at a bare minimum.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example - Deadlock with Two Tasks**

```
void task1(void);
void task2(void);

int var;
void perform_task_cycle(void) {
 var++;
}

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2(); //Noncompliant
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_2();
    begin_critical_section_1();
    perform_task_cycle();
    end_critical_section_1();
    end_critical_section_2();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** | ☑ | |
| **Entry points** | task1<br>task2 | |
| **Critical section details** | **Starting routine** | **Ending routine** |
| | begin_critical_section_1 | end_critical_section_1 |
| | begin_critical_section_2 | end_critical_section_2 |

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `begin_critical_section_1`.
2. `task2` calls `begin_critical_section_2`.
3. `task1` reaches the instruction `begin_critical_section_2();`. Since `task2` has already called `begin_critical_section_2`, `task1` waits for `task2` to call `end_critical_section_2`.
4. `task2` reaches the instruction `begin_critical_section_1();`. Since `task1` has already called `begin_critical_section_1`, `task2` waits for `task1` to call `end_critical_section_1`.

#### Correction-Follow Same Locking Sequence in Both Tasks

One possible correction is to follow the same sequence of calls to lock and unlock functions in both `task1` and `task2`.

```
void task1(void);
void task2(void);
void perform_task_cycle(void);

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}
```

**Example - Deadlock with More Than Two Tasks**

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);


void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2(); //Noncompliant
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock3();
    lock1();
    performTaskCycle();
    unlock1();
    unlock3();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** | ☑ | |
| **Entry points** | `task1`<br>`task2`<br>`task3` | |
| **Critical section details** | **Starting routine** | **Ending routine** |
| | `lock1` | `unlock1` |
| | `lock2` | `unlock2` |
| | `lock3` | `unlock3` |

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `lock1`.
2. `task2` calls `lock2`.
3. `task3` calls `lock3`.
4. `task1` reaches the instruction `lock2();`. Since `task2` has already called `lock2`, `task1` waits for call to `unlock2`.
5. `task2` reaches the instruction `lock3();`. Since `task3` has already called `lock3`, `task2` waits for call to `unlock3`.
6. `task3` reaches the instruction `lock1();`. Since `task1` has already called `lock1`, `task3` waits for call to `unlock1`.

#### Correction — Break Cyclic Order

To break the cyclic order between critical sections, note every lock function in your code in a certain sequence, for example:

1. `lock1`
2. `lock2`
3. `lock3`

If you use more than one lock function in a task, use them in the order in which they appear in the sequence. For example, you can use `lock1` followed by `lock2` but not `lock2` followed by `lock1`.

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);

void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock1();
    lock3();
    performTaskCycle();
    unlock3();
    unlock1();
 }
}
```

## Check Information

**Group:** Rule 50. POSIX (POS)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

POS51-C

# CERT C: Rule POS52-C

Do not perform operations that can block while holding a POSIX lock

## Description

### Rule Definition

*Do not perform operations that can block while holding a POSIX lock.*

### Polyspace Implementation

The rule checker checks for **Blocking operation while holding lock**.

## Examples

### Blocking operation while holding lock

#### Issue

**Blocking operation while holding lock** occurs when a task (thread) performs a potentially lengthy operation while holding a lock.

The checker considers calls to these functions as potentially lengthy:

- Functions that access a network such as `recv`
- System call functions such as `fork`, `pipe` and `system`
- Functions for I/O operations such as `getchar` and `scanf`
- File handling functions such as `fopen`, `remove` and `lstat`
- Directory manipulation functions such as `mkdir` and `rmdir`

The checker automatically detects certain primitives that hold and release a lock, for instance, `pthread_mutex_lock` and `pthread_mutex_unlock`. For the full list of primitives that are automatically detected, see “Auto-Detection of Thread Creation and Critical Section in Polyspace”.

#### Risk

If a thread performs a lengthy operation when holding a lock, other threads that use the lock have to wait for the lock to be available. As a result, system performance can slow down or deadlocks can occur.

#### Fix

Perform the blocking operation before holding the lock or after releasing the lock.

Some functions detected by this checker can be called in a way that does not make them potentially lengthy. For instance, the function `recv` can be called with the parameter `O_NONBLOCK` which causes the call to fail if no message is available. When called with this parameter, `recv` does not wait for a message to become available.

#### Example - Network I/O Operations with `recv` While Holding Lock

```
#include <pthread.h>
#include <sys/socket.h>
```

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void thread_foo(void *ptr) {
  unsigned int num;
  int result;
  int sock;

  /* sock is a connected TCP socket */

  if ((result = pthread_mutex_lock(&mutex)) != 0) {
    /* Handle Error */
  }

  if ((result = recv(sock, (void *)&num, sizeof(unsigned int), 0)) < 0) { //Noncompliant
    /* Handle Error */
  }

  /* ... */

  if ((result = pthread_mutex_unlock(&mutex)) != 0) {
    /* Handle Error */
  }
}

int main() {
  pthread_t thread;
  int result;

  if ((result = pthread_mutexattr_settype(
      &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_init(&mutex, &attr)) != 0) {
    /* Handle Error */
  }

  if (pthread_create(&thread, NULL,(void*(*)(void*))& thread_foo, NULL) != 0) {
    /* Handle Error */
  }

  /* ... */

  pthread_join(thread, NULL);

  if ((result = pthread_mutex_destroy(&mutex)) != 0) {
    /* Handle Error */
  }

  return 0;
}
```

In this example, in each thread created with `pthread_create`, the function `thread_foo` performs a network I/O operation with `recv` after acquiring a lock with `pthread_mutex_lock`. Other threads using the same lock variable mutex have to wait for the operation to complete and the lock to become available.

### Correction — Perform Blocking Operation Before Acquiring Lock

One possible correction is to call recv before acquiring the lock.

```
#include <pthread.h>
#include <sys/socket.h>

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void thread_foo(void *ptr) {
  unsigned int num;
  int result;
  int sock;

  /* sock is a connected TCP socket */
  if ((result = recv(sock, (void *)&num, sizeof(unsigned int), 0)) < 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_lock(&mutex)) != 0) {
    /* Handle Error */
  }

    /* ... */

  if ((result = pthread_mutex_unlock(&mutex)) != 0) {
    /* Handle Error */
  }
}

int main() {
  pthread_t thread;
  int result;

  if ((result = pthread_mutexattr_settype(
      &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_init(&mutex, &attr)) != 0) {
    /* Handle Error */
  }

  if (pthread_create(&thread, NULL,(void*(*)(void*))& thread_foo, NULL) != 0) {
    /* Handle Error */
  }

  /* ... */

  pthread_join(thread, NULL);

  if ((result = pthread_mutex_destroy(&mutex)) != 0) {
    /* Handle Error */
  }
  return 0;
}
```

## Check Information

**Group:** Rule 50. POSIX (POS)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

POS52-C

# CERT C: Rule. POS53-C

Do not use more than one mutex for concurrent waiting operations on a condition variable

## Description

### Rule Definition

*Do not use more than one mutex for concurrent waiting operations on a condition variable.*

### Polyspace Implementation

The rule checker checks for **Multiple mutexes used with same conditional variable**.

## Examples

### Multiple mutexes used with same conditional variable

#### Issue

This issue occurs when multiple threads use more than one mutex to concurrently wait on the same condition variable. A thread waits on a condition variable by calling the functions `pthread_cond_timedwait` or `pthread_cond_wait`. These functions take a condition variable and a locked mutex as arguments, and the condition variable is bound to that mutex when the thread waits on the condition variable.

The checkers flags the use of `pthread_cond_timedwait` or `pthread_cond_wait` in one of the threads. See the **Event** column in the **Results Details** pane to view the threads waiting on the same condition variable and using a different mutex.

#### Risk

When a thread waits on a condition variable using a mutex, the condition variable is bound to that mutex. Any other thread using a different mutex to wait on the same condition variable is undefined behavior according to the POSIX standard.

#### Fix

Use the same mutex argument for `pthread_cond_timedwait` or `pthread_cond_wait` when threads are concurrently waiting on the same condition variable, or use separate condition variables for each mutex.

#### Example - Concurrent Waiting on Condition Variable with Multiple Mutexes

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define Thrd_return_t                    void *
#define __USE_XOPEN2K8


#define COUNT_LIMIT 5

static void fatal_error(void)
{
    exit(1);
}
```

```
pthread_mutex_t mutex1;
pthread_mutex_t mutex2;
pthread_mutex_t mutex3;
pthread_cond_t cv;

int count1 = 0, count2 = 0, count3 = 0;
#define DELAY 8

Thrd_return_t waiter1(void* arg)
{
    int ret;
    while (count1 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex1)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                 pthread_cond_wait(&cv, &mutex1)) != 0) { //Noncompliant
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count1 = %d\n", ++count1);
        if ((ret = pthread_mutex_unlock(&mutex1)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter2(void* arg)
{
    int ret;
    while (count2 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex2)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                 pthread_cond_wait(&cv, &mutex2)) != 0) { //Noncompliant
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count2 = %d\n", ++count2);
        if ((ret = pthread_mutex_unlock(&mutex2)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t signaler(void* arg)
{
    int ret;
    while ((count1 < COUNT_LIMIT) || (count2 < COUNT_LIMIT)) {
        sleep(1);
        printf("signaling\n");
        if ((ret = pthread_cond_broadcast(&cv)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter3(void* arg)
{
    int ret;
    while (count3 % COUNT_LIMIT != 0) {
        if ((ret = pthread_mutex_lock(&mutex3)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                 pthread_cond_wait(&cv, &mutex3)) != 0) { //Noncompliant
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count3 = %d\n", ++count3);
        if ((ret = pthread_mutex_unlock(&mutex3)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

int main(void)
{
    int ret;
    pthread_t thread1, thread2, thread3;

    pthread_mutexattr_t attr;

    if ((ret = pthread_mutexattr_init(&attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
        /* Handle error */
        fatal_error();
    }

    if ((ret = pthread_mutex_init(&mutex1, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex2, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex3, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_cond_init(&cv, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread1, NULL, &waiter1, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread2, NULL, &waiter2, NULL))) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread3, NULL, &signaler, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread1, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread2, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread3, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }

    while (1) { ; }

    return 0;
}
```

In this example, a different mutex is used to protect each `count` variable. Since all three `waiter` functions wait on the same condition variable `cv` with different mutexes, the call to `pthread_cond_wait` will succeed for one of the threads and the call will be undefined for the other two.

The checker raises a defect for function `waiter3` even though the function is not invoked directly or indirectly by a thread, entry-point, or interrupt. The analysis considers function `waiter3` called by the main program through its function address or an unidentified thread whose creation is the missing source code.

#### Correction — Use the Same Mutex for All Threads Waiting on Same Condition Variable

One possible correction is to pass the same mutex argument to all the call to `pthread_cond_wait` that are used to wait on the same condition variable.

```
 #include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define Thrd_return_t                    void *
#define __USE_XOPEN2K8



#define COUNT_LIMIT 5

static void fatal_error(void)
{
    exit(1);
}


pthread_mutex_t mutex;

pthread_cond_t cv;

int count1 = 0, count2 = 0, count3 = 0;
#define DELAY 8

Thrd_return_t waiter1(void* arg)
{
    int ret;
    while (count1 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                 pthread_cond_wait(&cv, &mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count1 = %d\n", ++count1);
        if ((ret = pthread_mutex_unlock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter2(void* arg)
{
    int ret;
    while (count2 < COUNT_LIMIT) {
        if ((ret = pthread_mutex_lock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                 pthread_cond_wait(&cv, &mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count2 = %d\n", ++count2);
        if ((ret = pthread_mutex_unlock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
```

```
}

Thrd_return_t signaler(void* arg)
{
    int ret;
    while ((count1 < COUNT_LIMIT) || (count2 < COUNT_LIMIT)) {
        sleep(1);
        printf("signaling\n");
        if ((ret = pthread_cond_broadcast(&cv)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}

Thrd_return_t waiter3(void* arg)
{
    int ret;
    while (count3 % COUNT_LIMIT != 0) {
        if ((ret = pthread_mutex_lock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        if ((ret =
                 pthread_cond_wait(&cv, &mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
        sleep(random() % DELAY);
        printf("count3 = %d\n", ++count3);
        if ((ret = pthread_mutex_unlock(&mutex)) != 0) {
            /* Handle error */
            fatal_error();
        }
    }
    return (Thrd_return_t)0;
}
/*
void user_task(void)
{
    (void)waiter3(NULL);
} */

int main(void)
{
    int ret;
    pthread_t thread1, thread2, thread3;

    pthread_mutexattr_t attr;

    if ((ret = pthread_mutexattr_init(&attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
        /* Handle error */
        fatal_error();
    }

    if ((ret = pthread_mutex_init(&mutex, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_mutex_init(&mutex, &attr)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_cond_init(&cv, NULL)) != 0) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread1, NULL, &waiter1, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread2, NULL, &waiter2, NULL))) {
        /* handle error */
        fatal_error();
    }
    if ((ret = pthread_create(&thread3, NULL, &signaler, NULL))) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread1, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread2, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }
    if ((ret = pthread_join(thread3, NULL)) != 0) {
        /* Handle error */
        fatal_error();
    }

    while (1) { ; }

    return 0;
}
```

## Check Information

**Group:** Rule 50. POSIX (POS)

# Version History

**Introduced in R2020a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

POS53-C

# CERT C: Rule POS54-C

Detect and handle POSIX library errors

## Description

### Rule Definition

*Detect and handle POSIX library errors.*

### Polyspace Implementation

The rule checker checks for **Returned value of a sensitive POSIX function not checked**.

## Examples

### Returned value of a sensitive POSIX function not checked

#### Issue

**Returned value of a sensitive POSIX function not checked** occurs when you call sensitive POSIX functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

#### Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

#### Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

#### Example - Sensitive Function Return Ignored

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);//Noncompliant
}
int read_file(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r");
  return 0;  //Noncompliant

}
```

This example shows calls to the sensitive POSIX functions `pthread_attr_init` and `fmemopen`. Their return values are ignored, causing defect.

#### Correction — Cast Function to (void)

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);//Compliant
}
int read_file(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  (void)fmemopen (argv[1], strlen (argv[1]), "r"); //Compliant
  return 0;
}
```

**Correction — Test Return Value**

One possible correction is to test the return value of `pthread_attr_init` and `fmemopen` to check for errors.

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

void initialize() {
    pthread_attr_t attr;

    int result = pthread_attr_init(&attr);//Compliant
    if(result != 0){
        //Handle fatal error
    }
}
int read_file(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r"); //Compliant
  if (in==NULL){
      // Handle error
  }
  return 0;
}
```

**Example - Critical Function Return Ignored**

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0)); //Noncompliant
    pthread_join(thread_id,  &res);  //Noncompliant
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

#### Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

## Check Information

**Group:** Rule 50. POSIX (POS)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

POS54-C

# CERT C: Rule WIN30-C

Properly pair allocation and deallocation functions

## Description

### Rule Definition

*Properly pair allocation and deallocation functions.*

### Polyspace Implementation

The rule checker checks for **Mismatched alloc/dealloc functions on Windows**.

## Examples

### Mismatched alloc/dealloc functions on Windows

#### Issue

**Mismatched alloc/dealloc functions on Windows** occurs when you use a Windows deallocation function that is not properly paired to its corresponding allocation function.

#### Risk

Deallocating memory with a function that does not match the allocation function can cause memory corruption or undefined behavior. If you are using an older version of Windows, the improper function can also cause compatibility issues with newer versions.

#### Fix

Properly pair your allocation and deallocation functions according to the functions listed in this table.

| Allocation Function | Deallocation Function |
|---|---|
| `malloc()` | `free()` |
| `realloc()` | `free()` |
| `calloc()` | `free()` |
| `_aligned_malloc()` | `_aligned_free()` |
| `_aligned_offset_malloc()` | `_aligned_free()` |
| `_aligned_realloc()` | `_aligned_free()` |
| `_aligned_offset_realloc()` | `_aligned_free()` |
| `_aligned_recalloc()` | `_aligned_free()` |
| `_aligned_offset_recalloc()` | `_aligned_free()` |
| `_malloca()` | `_freea()` |
| `LocalAlloc()` | `LocalFree()` |
| `LocalReAlloc()` | `LocalFree()` |
| `GlobalAlloc()` | `GlobalFree()` |

| Allocation Function | Deallocation Function |
|---|---|
| GlobalReAlloc() | GlobalFree() |
| VirtualAlloc() | VirtualFree() |
| VirtualAllocEx() | VirtualFreeEx() |
| VirtualAllocExNuma() | VirtualFreeEx() |
| HeapAlloc() | HeapFree() |
| HeapReAlloc() | HeapFree() |

**Example - Memory Deallocated with Incorrect Function**

```
#ifdef _WIN32_
#include <windows.h>
#else
#define _WIN32_
typedef void *HANDLE;
typedef HANDLE HGLOBAL;
typedef HANDLE HLOCAL;
typedef unsigned int UINT;
extern HLOCAL LocalAlloc(UINT uFlags, UINT uBytes);
extern HLOCAL LocalFree(HLOCAL hMem);
extern HGLOBAL GlobalFree(HGLOBAL hMem);
#endif

#define SIZE9 9


void func(void)
{
    /* Memory allocation */
    HLOCAL p = LocalAlloc(0x0000, SIZE9);

    if (p) {
        /* Memory deallocation. */
        GlobalFree(p); //Noncompliant

    }
}
```

In this example, memory is allocated with `LocallAlloc()`. The program then erroneously uses `GlobalFree()` to deallocate the memory.

**Correction — Properly Pair Windows Allocation and Deallocation Functions**

When you allocate memory with `LocalAllocate()`, use `LocalFree()` to deallocate the memory.

```
#ifdef _WIN32_
#include <windows.h>
#else
#define _WIN32_
typedef void *HANDLE;
typedef HANDLE HGLOBAL;
typedef HANDLE HLOCAL;
typedef unsigned int UINT;
```

```
extern HLOCAL LocalAlloc(UINT uFlags, UINT uBytes);
extern HLOCAL LocalFree(HLOCAL hMem);
extern HGLOBAL GlobalFree(HGLOBAL hMem);
#endif

#define SIZE9 9
void func(void)
{
    /* Memory allocation */
    HLOCAL p = LocalAlloc(0x0000, SIZE9);
    if (p) {
        /* Memory deallocation. */
        LocalFree(p);
    }
}
```

## Check Information

**Group:** Rule 51. Microsoft Windows (WIN)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

WIN30-C

# CERT C: Rec. PRE00-C

Prefer inline or static functions to function-like macros

## Description

### Rule Definition

*Prefer inline or static functions to function-like macros.*

### Polyspace Implementation

The rule checker checks for **Use of function-like macro instead of function**.

## Examples

### Use of function-like macro instead of function

#### Issue

The issue occurs when you use a function-like macro instead of a function when the two are interchangeable.

Polyspace considers all function-like macro definitions.

#### Risk

In most circumstances, use functions instead of macros. Functions perform argument type-checking and evaluate their arguments once, avoiding problems with potential multiple side effects.

## Check Information

**Group:** Rec. 01. Preprocessor (PRE)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

PRE00-C

# CERT C: Rec. PRE01-C

Use parentheses within macros around parameter names

## Description

### Rule Definition

*Use parentheses within macros around parameter names.*

### Polyspace Implementation

The rule checker checks for **Macro parameters not enclosed in parentheses**.

## Examples

### Macro parameters not enclosed in parentheses

**Issue**

The issue occurs when a macro parameter contains an expression but you do not enclose the parameter in parentheses either in the macro definition or in the macro use.

If a macro parameter does not contain an expression, then the parentheses are not necessary.

**Risk**

If you do not enclose macro parameters containing expressions in parentheses, when parameter substitution occurs, operator precedence might not give the results that you want .

**Fix**

If a macro parameter contains an expression, enclose the parameter in parenthesis in the macro definition or macro use.

**Example - Macro Expressions**

```
#define mac1(x, y) (x * y)
#define mac2(x, y) ((x) * (y))

void foo(void){
    int r;

    r = mac1(1 + 2, 3 + 4);       /* Non-compliant */
    r = mac1((1 + 2), (3 + 4));   /* Compliant */

    r = mac2(1 + 2, 3 + 4);       /* Compliant */
}
```

In this example, `mac1` and `mac2` are macros that presumably implement the same definition.

- The definition of `mac1` does not enclose the macro parameters in parentheses. In the noncompliant expression, the macro expands to `r = (1 + 2 * 3 + 4);` The intended expression can be `(1 + (2 * 3) + 4)` or `(1 + 2) * (3 + 4)`. However, without parentheses, a developer or code reviewer might not know the intent of the expression. The subsequent compliant expression encloses the macro parameters in parentheses, so the intended macro expansion is clearly `(1 + 2) * (3 + 4)`.

- The definition of `mac2` encloses the parameters in parentheses. The expression involving `mac2` expands to `(1 + 2) * (3 + 4)` and complies with the rule.

## Check Information

**Group:** Rec. 01. Preprocessor (PRE)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

PRE01-C

# CERT C: Rec. PRE06-C

Enclose header files in an inclusion guard

## Description

### Rule Definition

*Enclose header files in an inclusion guard.*

### Polyspace Implementation

The rule checker checks for **Contents of header file not guarded from multiple inclusions**.

## Examples

### Contents of header file not guarded from multiple inclusions

#### Issue

The issue occurs when you do not take precautions order to prevent the contents of a header file being included more than once.

If you include a header file whose contents are not guarded from multiple inclusion, the analysis raises a violation of this directive. The violation is shown at the beginning of the header file.

You can guard the contents of a header file from multiple inclusion by using one of the following methods:

```
//<start-of-file>
#ifndef <control macro>
#define <control macro>
    /* Contents of file */
#endif
//<end-of-file>
```

or

```
//<start-of-file>
#ifdef <control macro>
#error ...
#else
#define <control macro>
    /* Contents of file */
#endif
//<end-of-file>
```

Unless you use one of these methods, Polyspace flags the header file inclusion as noncompliant.

#### Risk

When a translation unit contains a complex hierarchy of nested header files, it is possible for a particular header file to be included more than once, leading to confusion. If this multiple inclusion produces multiple or conflicting definitions, then your program can have undefined or erroneous behavior.

For instance, suppose that a header file contains:

```
#ifdef _WIN64
   int env_var;
#elseif
   long int env_var;
#endif
```

If the header file is contained in two inclusion paths, one that defines the macro `_WIN64` and another that undefines it, you can have conflicting definitions of `env_var`.

### Example - Code After Macro Guard

```
#ifndef  __MY_MACRO__
#define __MY_MACRO__
    void func(void);
#endif
void func2(void);
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func2(void)` is outside the guard.

**Note** You can have comments outside the macro guard.

### Example - Code Before Macro Guard

```
void func(void);
#ifndef  __MY_MACRO__
#define __MY_MACRO__
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func(void)` is outside the guard.

**Note** You can have comments outside the macro guard.

### Example - Mismatch in Macro Guard

```
#ifndef  __MY_MACRO__
#define __MY_MARCO__
    void func(void);
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro name in the `#ifndef` statement is different from the name in the following `#define` statement.

## Check Information

**Group:** Rec. 01. Preprocessor (PRE)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

PRE06-C

# CERT C: Rec. PRE07-C

Avoid using repeated question marks

## Description

### Rule Definition

*Avoid using repeated question marks.*

### Polyspace Implementation

The rule checker checks for **Use of trigraphs**.

## Examples

### Use of trigraphs

#### Issue

The issue occurs when you use trigraphs in your code.

The Polyspace analysis converts trigraphs to the equivalent character for the defect analysis. However, Polyspace also raises a MISRA violation.

The standard requires that trigraphs must be transformed *before* comments are removed during preprocessing. Therefore, Polyspace raises a violation of this rule even if a trigraph appears in code comments.

#### Risk

You denote trigraphs with two question marks followed by a specific third character (for instance,`'??-'` represents a `'~'` (tilde) character and `'??)'` represents a `']'`). These trigraphs can cause accidental confusion with other uses of two question marks.

**Note** Digraphs (`<:  :>`, `<%  %>`, `%:`, `%:%:`) are permitted because they are tokens.

## Check Information

**Group:** Rec. 01. Preprocessor (PRE)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**
PRE07-C

# CERT C: Rec. PRE09-C

Do not replace secure functions with deprecated or obsolescent functions

## Description

### Rule Definition

*Do not replace secure functions with deprecated or obsolescent functions.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Use of dangerous standard function**.
- **Insufficient destination buffer size**.

## Examples

### Use of dangerous standard function

#### Issue

The **Use of dangerous standard function** check highlights uses of functions that are inherently dangerous or potentially dangerous given certain circumstances. The following table lists possibly dangerous functions, the risks of using each function, and what function to use instead.

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `gets` | Inherently dangerous — You cannot control the length of input from the console. | `fgets` |
| `cin` | Inherently dangerous — You cannot control the length of input from the console. | Avoid or prefaces calls to `cin` with `cin.width`. |
| `strcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `strncpy` |
| `stpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `stpncpy` |
| `lstrcpy` or `StrCpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy` |
| `strcat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `strncat`, `strlcat`, or `strcat_s` |

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `lstrcat` or `StrCat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat` |
| `wcpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcpncpy` |
| `wcscat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `wcsncat`, `wcslcat`, or `wcncat_s` |
| `wcscpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcsncpy` |
| `sprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `snprintf` |
| `vsprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `vsnprintf` |

#### Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Using `sprintf`

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
```

```
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1) //Noncompliant
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

**Correction — Use `snprintf` with Buffer Size**

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

### Insufficient destination buffer size

**Issue**

**Insufficient destination buffer size** occurs when the destination buffer in a `strcpy` operation cannot accommodate the source buffer and a null terminator. This issue is reported if the size of the source buffer is unknown. Consider this code:

```
int main (int argc, char *argv[])
{
  const char *const input = ((argc && argv[0]) ? argv[0] : "");
  char str[100];
  strcpy(input, str); // Noncompliant
}
```

In this case, the size of the source buffer `input` is unknown. The size of the destination buffer `str` might be smaller than the value `(strlen(input)+1)`. Polyspace reports a violation on the `strcpy` operation.

#### Risk

Using a destination buffer of insufficient size might allow an attacker to cause a buffer overflow. In the preceding code example, if `argv[0]` contains 100 or more characters, the `strcpy` operation results in a buffer overflow.

#### Fix

Before calling the function `strcpy()`, allocate sufficient memory dynamically. For instance, use the function `strlen()` to determine the size of the source buffer and then allocate the destination buffer so that its size is greater than the value `strlen(source) + 1`.

#### Example — Destination Buffer Too Small

In this example, the size of the `source` buffer is unknown, while the size of the `destination` buffer is fixed at `128`. The size of the `destination` buffer might not be sufficient to accommodate the characters from the `source` buffer and terminate the buffer with a null. Polyspace reports a violation of the rule.

```
#include <string.h>

int main(int argc, char *argv[]) {
  const char *const source = (argc && argv[0]) ? argv[0] : "";
  char destination[128];
  strcpy(source, destination);//Noncompliant

  return 0;
}
```

#### Correction — Allocate Sufficient Memory for Destination Buffer

This violation is resolved by allocating sufficient memory for the `destination` buffer. For instance, use the function `strlen()` to calculate the size of the `source` buffer and allocate sufficient memory for the `destination` buffer so that it can accommodate all characters from the `source` buffer and the null terminator (`'\0'` ).

```
#include <string.h>

int main(int argc, char *argv[]) {
  const char *const source = (argc && argv[0]) ? argv[0] : "";
  char* destination = (char *)malloc(strlen(source)+ 1);
  if(destination!=NULL){
      strcpy(source, destination);//Compliant
  }else{
      /*Handle Error*/
  }
  //...
  free(destination);
  return 0;
}
```

## Check Information

**Group:** Rec. 01. Preprocessor (PRE)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

PRE09-C

# CERT C: Rec. PRE10-C

Wrap multistatement macros in a do-while loop

## Description

### Rule Definition

*Wrap multistatement macros in a do-while loop.*

### Polyspace Implementation

The rule checker checks for **Macro with multiple statements**.

## Examples

### Macro with multiple statements

#### Issue

**Macro with multiple statements** occurs when a macro contains multiple semicolon-terminated statements, irrespective of whether the statements are enclosed in braces.

#### Risk

The macro expansion, in certain contexts such as an `if` condition or a loop, can lead to unintended program logic.

For instance, consider the macro:

```
#define RESET(x,y) \
   x=0; \
   y=0;
```

In an `if` statement such as:

```
if(checkSomeCondition)
   RESET(x,y);
```

the macro expands to:

```
if(checkSomething)
   x=0;
y=0;
```

which might be unexpected if you want both statements to be executed in an `if` block.

#### Fix

In a macro definition, wrap multiple statements in a `do...while(0)` loop.

For instance, in the preceding example, use the definition:

```
#define RESET(x,y) \
   do { \
     x=0; \
     y=0; \
   } while(0)
```

This macro is appropriate to expand in all contexts. The `while(0)` ensures that the statements are executed only once.

Alternatively, use inline functions in preference to function-like macros that involve multiple statements.

Note that the loop is required for the correct solution and wrapping the statements in braces alone does not fix the issue. The macro expansion can still lead to unintended code.

#### Example – Macro with Multiple Statements

```
#define RESET(x,y) \ //Noncompliant
   x=0; \
   y=0;

void func(int *x, int *y, int resetFlag){
    if(resetFlag)
        RESET(x,y);
}
```

In this example, the defect occurs because the macro `RESET` consists of multiple statements.

#### Correction – Wrap Multiple Statements of Macro in do-while Loop

Wrap the statements of the macro in a `do..while(0)` loop in the macro definition.

```
#define RESET(x,y) \
   do { \
     x=0; \
     y=0; \
   } while(0)

void func(int *x, int *y, int resetFlag){
    if(resetFlag)
        RESET(x,y);
}
```

## Check Information

**Group:** Rec. 01. Preprocessor (PRE)

# Version History

**Introduced in R2020a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

PRE10-C

# CERT C: Rec. PRE11-C

Do not conclude macro definitions with a semicolon

## Description

### Rule Definition

*Do not conclude macro definitions with a semicolon.*

### Polyspace Implementation

The rule checker checks for **Macro terminated with a semicolon**.

## Examples

### Macro terminated with a semicolon

#### Issue

**Macro terminated with a semicolon** occurs when a macro that is invoked at least once has a definition ending with a semicolon.

#### Risk

If a macro definition ends with a semicolon, the macro expansion can lead to unintended program logic in certain contexts, such as within an expression.

For instance, consider the macro:

```
#define INC_BY_ONE(x) ++x;
```

If used in the expression:

```
res = INC_BY_ONE(x)%2;
```

the expression resolves to:

```
res = ++x; %2;
```

The value of `x+1` is assigned to `res`, which is probably unintended. The leftover standalone statement `%2;` is valid C code and can only be detected by enabling strict compiler warnings.

#### Fix

Do not end macro definitions with a semicolon. Leave it up to users of the macro to add a semicolon after the macro when needed.

Alternatively, use inline functions in preference to function-like macros that involve statements ending with semicolon.

#### Example – Spurious Semicolon in Macro Definition

```
#define WHILE_LOOP(n) while(n>0); //Noncompliant
```

```
void performAction(int timeStep);

void main() {
    int loopIter = 100;
    WHILE_LOOP(loopIter) {
        performAction(loopIter);
        loopIter--;
    }
}
```

In this example, the defect occurs because the definition of the macro `WHILE_LOOP(n)` ends with a semicolon. As a result of the semicolon, the `while` loop has an empty body and the following statements run only once. It was probably intended that the loop must run 100 times.

**Correction – Remove Semicolon from Macro Definition**

Remove the trailing semicolon from the macro definition. Users of the macro can add a semicolon after the macro when needed. In this example, a semicolon is not required.

```
#define WHILE_LOOP(n) while(n>0)

void performAction(int timeStep);

void main() {
    int loopIter = 100;
    WHILE_LOOP(loopIter) {
        performAction(loopIter);
        loopIter--;
    }
}
```

## Check Information

**Group:** Rec. 01. Preprocessor (PRE)

# Version History

**Introduced in R2020a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

PRE11-C

# CERT C: Rec. DCL00-C

Const-qualify immutable objects

## Description

### Rule Definition

*Const-qualify immutable objects.*

### Polyspace Implementation

The rule checker checks for **Unmodified variable not const-qualified**.

## Examples

### Unmodified variable not const-qualified

#### Issue

**Unmodified variable not const-qualified** occurs when a local variable is not `const`-qualified and one of the following statements is true during the variable lifetime:

- You do not perform write operations on the variable after initialization.
- When you perform write operations, you reassign the same constant value to the variable.

The checker considers a variable as modified if its address is assigned to a pointer or reference (unless it is a pointer or reference to a `const` variable), passed to another function, or otherwise used. In these situations, the checker does not suggest adding a `const` qualifier.

The checker flags arrays as candidates for `const`-qualification only if you do not perform write operations on the array elements at all after initialization.

#### Risk

`const`-qualifying a variable avoids unintended modification of the variable during later code maintenance. The `const` qualifier also indicates to a developer that the variable retains its initial value in the remainder of the code.

#### Fix

If you do not expect to modify a variable value during its lifetime, add the `const` qualifier to the variable declaration and initialize the variable at declaration.

If you expect the variable to be modified, see if the absence of a modification indicates a programming omission and fix the issue.

#### Example - Missing `const` Qualification on Pointer

```
#include <string.h>

char returnNthCharacter (int n) {
    char* pwd = "aXeWdf10fg" ; //Noncompliant
    char nthCharacter;
    for(int i=0; i < strlen(pwd); i++) {
        if(i==n)
            nthCharacter = pwd[i];
    }
    return nthCharacter;
}
```

In this example, the pointer pwd is not const-qualified. However, beyond initialization with a constant, it is not reassigned anywhere in the returnNthCharacter function.

#### Correction – Add const at Variable Declaration

If the variable is not intended to be modified, add the const qualifier at declaration. In this example, both the pointer and the pointed variable are not modified. Add a const qualifier to both the pointer and the pointed variable. Later modifications cannot reassign the pointer pwd to point at a different variable nor modify the value at the pointed location.

```
#include <string.h>

char returnNthCharacter (int n) {
    const char* const pwd = "aXeWdf10fg" ; //Compliant
    char nthCharacter;

    for(int i=0; i < strlen(pwd); i++) {
        if(i==n)
            nthCharacter = pwd[i];
    }
    return nthCharacter;
}
```

Note that the checker only flags the missing const from the pointer declaration. The checker does not determine if the pointed location also merits a const qualifier.

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2020b**

## See Also

Check SEI CERT-C (-cert-c)

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL00-C

# CERT C: Rec. DCL01-C

Do not reuse variable names in subscopes

## Description

### Rule Definition

*Do not reuse variable names in subscopes.*

### Polyspace Implementation

The rule checker checks for **Variable shadowing**.

## Examples

### Variable shadowing

#### Issue

**Variable shadowing** occurs when a variable hides another variable of the same name in an outer scope.

For instance, if a local variable has the same name as a global variable, the local variable hides the global variable during its lifetime.

#### Risk

When two variables with the same name exist in an inner and outer scope, any reference to the variable name uses the variable in the inner scope. However, a developer or reviewer might incorrectly expect that the variable in the outer scope was used.

#### Fix

The fix depends on the root cause of the defect. For instance, suppose you refactor a function such that you use a local static variable in place of a global variable. In this case, the global variable is redundant and you can remove its declaration. Alternatively, if you are not sure if the global variable is used elsewhere, you can modify the name of the local static variable and all references within the function.

If the shadowing is intended and you do not want to fix the issue, add comments to your result or code to avoid another review. See

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Variable Shadowing Error

```
#include <stdio.h>
```

```
int fact[5]={1,2,6,24,120};

int factorial(int n)
 {
  int fact=1;  //Noncompliant
  /*Defect: Local variable hides global array with same name */

  for(int i=1;i<=n;i++)
    fact*=i;

  return(fact);
 }
```

Inside the `factorial` function, the integer variable `fact` hides the global integer array `fact`.

**Correction — Change Variable Name**

One possible correction is to change the name of one of the variables, preferably the one with more local scope.

```
#include <stdio.h>

int fact[5]={1,2,6,24,120};

int factorial(int n)
 {
  /* Fix: Change name of local variable */
  int f=1;

  for(int i=1;i<=n;i++)
    f*=i;

  return(f);
 }
```

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL01-C

# CERT C: Rec. DCL02-C

Use visually distinct identifiers

## Description

### Rule Definition

*Use visually distinct identifiers.*

### Polyspace Implementation

The rule checker checks for **Use of typographically ambiguous identifiers**.

## Examples

### Use of typographically ambiguous identifiers

#### Issue

The issue occurs when you use identifiers in the same name space with overlapping visibility and the identifiers are not typographically unambiguous.

#### Risk

What “unambiguous” means depends on the alphabet and language in which source code is written. When you use identifiers that are typographically close, you can confuse between them.

For the Latin alphabet as used in English words, at a minimum, the identifiers should not differ by:

- The interchange of a lowercase letter with its uppercase equivalent.
- The presence or absence of the underscore character.
- The interchange of the letter O and the digit 0.
- The interchange of the letter I and the digit 1.
- The interchange of the letter I and the letter l.
- The interchange of the letter S and the digit 5.
- The interchange of the letter Z and the digit 2.
- The interchange of the letter n and the letter h.
- The interchange of the letter B and the digit 8.
- The interchange of the letters rn and the letter m.

#### Example - Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val;  /* Non-compliant */

    int id2_numval;
    int id2_numVal;   /* Non-compliant */
    int id3_lvalue;
    int id3_Ivalue;   /* Non-compliant */

    int id4_xyZ;
    int id4_xy2;      /* Non-compliant */

    int id5_zerO;
    int id5_zer0;     /* Non-compliant */

    int id6_rn;
    int id6_m;        /* Non-compliant */
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL02-C

# CERT C: Rec. DCL06-C

Use meaningful symbolic constants to represent literal values

## Description

### Rule Definition

*Use meaningful symbolic constants to represent literal values.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Hard-coded buffer size**.
- **Hard-coded loop boundary**.

## Examples

### Hard-coded buffer size

#### Issue

**Hard-coded buffer size** occurs when you use a numerical value instead of a symbolic constant when declaring a memory buffer such as an array.

#### Risk

Hard-coded buffer size causes the following issues:

- Hard-coded buffer size increases the likelihood of mistakes and therefore maintenance costs. If a policy change requires developers to change the buffer size, they must change every occurrence of the buffer size in the code.
- Hard-constant constants can be exposed to attack if the code is disclosed.

#### Fix

Use a symbolic name instead of a hard-coded constant for buffer size. Symbolic names include `const`-qualified variables, `enum` constants, or macros.

`enum` constants are recommended.

- Macros are replaced by their constant values after preprocessing. Therefore, they can expose the loop boundary.
- `enum` constants are known at compilation time. Therefore, compilers can optimize the loops more efficiently.

  `const`-qualified variables are usually known at run time.

#### Example - Hard-Coded Buffer Size

```
int table[100]; //Noncompliant
```

```
void read(int);

void func(void) {
    for (int i=0; i<100; i++) //Noncompliant
        read(table[i]);
}
```

In this example, the size of the array `table` and the loop boundary in the `for` loop are hard-coded.

#### Correction — Use Symbolic Name

One possible correction is to replace the hard-coded size with a symbolic name.

```
const int MAX_1 = 100;
#define MAX_2 100
enum { MAX_3 = 100 };

int table_2[MAX_2];
int table_3[MAX_3];

void read(int);

void func(void) {
    int table_1[MAX_1];
    for (int i=0; i < MAX_1; i++)
        read(table_1[i]);
    for (int i=0; i < MAX_2; i++)
        read(table_2[i]);
    for (int i=0; i < MAX_3; i++)
        read(table_3[i]);
}
```

### Hard-coded loop boundary

#### Issue

**Hard-coded loop boundary** occurs when you use a numerical value instead of symbolic constant for the boundary of a `for`, `while` or `do-while` loop.

#### Risk

Hard-coded loop boundary causes the following issues:

- Hard-coded loop boundary makes the code vulnerable to denial of service attacks when the loop involves time-consuming computation or resource allocation.
- Hard-coded loop boundary increases the likelihood of mistakes and maintenance costs. If a policy change requires developers to change the loop boundary, they must change every occurrence of the boundary in the code.

  For instance, the loop boundary is 10000 and represents the maximum number of client connections supported in a network server application. If the server supports more clients, you must change all instances of the loop boundary in your code. Even if the loop boundary occurs once, you have to search for a numerical value of `10000` in your code. The numerical value can occur in places other than the loop boundary. You must browse through those places before you find the loop boundary.

#### Fix

Use a symbolic name instead of a hard-coded constant for loop boundary. Symbolic names include `const`-qualified variables, `enum` constants or macros.`enum` constants are recommended because:

- Macros are replaced by their constant values after preprocessing. Therefore, they can expose the buffer size.
- `enum` constants are known at compilation time. Therefore, compilers can allocate storage for them more efficiently.

  `const`-qualified variables are usually known at run time.

#### Example - Hard-Coded Loop Boundary

```
void performOperation(int);

void func(void) {
    for (int i=0; i<100; i++) //Noncompliant
        performOperation(i);
}
```

In this example, the boundary of the `for` loop is hard-coded.

#### Correction — Use Symbolic Name

One possible correction is to replace the hard-coded loop boundary with a symbolic name.

```
const int MAX_1 = 100;
#define MAX_2 100
enum { MAX_3 = 100 };

void performOperation_1(int);
void performOperation_2(int);
void performOperation_3(int);

void func(void) {
    for (int i=0; i<MAX_1; i++)
        performOperation_1(i);
    for (int i=0; i<MAX_2; i++)
        performOperation_2(i);
    for (int i=0; i<MAX_3; i++)
        performOperation_3(i);
}
```

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**
“Check for and Review Coding Standard Violations”

**External Websites**
DCL06-C

# CERT C: Rec. DCL07-C

Include the appropriate type information in function declarators

## Description

### Rule Definition

*Include the appropriate type information in function declarators.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Cast between function pointers with different types**.
- **Function declared implicitly**.

## Examples

### Cast between function pointers with different types

#### Issue

The issues occurs when you perform a conversion between a pointer to a function and any other type.

Polyspace considers both explicit and implicit casts when checking this rule. However, casts from `NULL` or `(void*)0` do not violate this rule.

#### Risk

The rule forbids the following two conversions:

- Conversion from a function pointer to any other type. This conversion causes undefined behavior.
- Conversion from a function pointer to another function pointer, if the function pointers have different argument and return types.

  The conversion is forbidden because calling a function through a pointer with incompatible type results in undefined behavior.

#### Example - Cast between two function pointers

```
typedef void (*fp16) (short n);
typedef void (*fp32) (int n);

#include <stdlib.h>                     /* To obtain macro  NULL */

void func(void) {   /* Exception 1 - Can convert a null pointer
                     * constant into a pointer to a function */
  fp16 fp1 = NULL;                 /* Compliant - exception  */
  fp16 fp2 = (fp16) fp1;           /* Compliant */
  fp32 fp3 = (fp32) fp1;           /* Non-compliant */
  if (fp2 != NULL) {}              /* Compliant - exception  */
  fp16 fp4 = (fp16) 0x8000;        /* Non-compliant - integer to
                                    * function pointer */}
```

In this example, the rule is violated when:

- The pointer `fp1` of type `fp16` is cast to type `fp32`. The function pointer types `fp16` and `fp32` have different argument types.
- An integer is cast to type `fp16`.

The rule is not violated when function pointers `fp1` and `fp2` are cast to `NULL`.

### Function declared implicitly

#### Issue

The issue occurs when you declare a function implicitly.

#### Risk

An implicit declaration occurs when you call a function before declaring or defining it. When you declare a function explicitly before calling it, the compiler can match the argument and return types with the parameter types in the declaration. If an implicit declaration occurs, the compiler makes assumptions about the argument and return types. For instance, it assumes a return type of `int`. The assumptions might not agree with what you expect and cause undesired type conversions.

#### Example - Function Not Declared Before Call

```
#include <math.h>

extern double power3 (double val, int exponent);
int getChoice(void);

double func() {
    double res;
    int ch = getChoice();
    if(ch == 0) {
        res = power(2.0, 10);    /* Non-compliant */
    }
    else if( ch==1) {
        res = power2(2.0, 10);   /* Non-compliant */
    }
    else {
        res = power3(2.0, 10);   /* Compliant */
        return res;
    }
}

double power2 (double val, int exponent) {
    return (pow(val, exponent));
}
```

In this example, the rule is violated when a function that is not declared is called in the code. Even if a function definition exists later in the code, the rule violation occurs.

The rule is not violated when the function is declared before it is called in the code. If the function definition exists in another file and is available only during the link phase, you can declare the function in one of the following ways:

- Declare the function with the `extern` keyword in the current file.
- Declare the function in a header file and include the header file in the current file.

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL07-C

# CERT C: Rec. DCL10-C

Maintain the contract between the writer and caller of variadic functions

## Description

### Rule Definition

*Maintain the contract between the writer and caller of variadic functions.*

### Polyspace Implementation

The rule checker checks for **Format string specifiers and arguments mismatch**.

## Examples

### Format string specifiers and arguments mismatch

#### Issue

**Format string specifiers and arguments mismatch** occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

#### Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

#### Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
const char *message = "License not available";
int err_number = -4;
printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {
    unsigned long fst = 1;

    printf("%d\n", fst); //Noncompliant
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

**Correction — Use an Unsigned Long Format Specifier**

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

**Correction — Use an Integer Argument**

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

DCL10-C

# CERT C: Rec. DCL11-C

Understand the type issues associated with variadic functions

## Description

### Rule Definition

*Understand the type issues associated with variadic functions.*

### Polyspace Implementation

The rule checker checks for **Format string specifiers and arguments mismatch**.

## Examples

### Format string specifiers and arguments mismatch

#### Issue

**Format string specifiers and arguments mismatch** occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

#### Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

#### Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
const char *message = "License not available";
int err_number = -4;
printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {
    unsigned long fst = 1;

    printf("%d\n", fst); //Noncompliant
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

#### Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

#### Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

### Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

DCL11-C

# CERT C: Rec. DCL12-C

Implement abstract data types using opaque types

## Description

### Rule Definition

*Implement abstract data types using opaque types.*

### Polyspace Implementation

The rule checker checks for **Structure or union object implementation visible in file where pointer to this object is not dereferenced**.

## Examples

### Structure or union object implementation visible in file where pointer to this object is not dereferenced

#### Issue

The issue occurs when a pointer to a structure or union is never dereferenced within a translation unit, but the implementation of the object is not hidden.

If a structure or union is defined in a file or a header file included in the file, a pointer to this structure or union declared but the pointer never dereferenced in the file, the checker flags a coding rule violation. The structure or union definition should not be visible to this file.

If you see a violation of this rule on a structure definition, identify if you have defined a pointer to the structure in the same file or in a header file included in the file. Then check if you dereference the pointer anywhere in the file. If you do not dereference the pointer, the structure definition should be hidden from this file and included header files.

`file.h`: Contains structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct  {
  int a;
} myStruct;

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

myStruct* getObj(void);
void useObj(myStruct*);

void func() {
  myStruct *sPtr = getObj();
  useObj(sPtr);
}
```

In this example, the pointer to the type `myStruct` is not dereferenced. The pointer is simply obtained from the `getObj` function and passed to the `useObj` function.

The implementation of `myStruct` is visible in the translation unit consisting of `file.c` and `file.h`.

One possible correction is to define an opaque data type in the header file `file.h`. The opaque data type `ptrMyStruct` points to the `myStruct` structure without revealing what the structure contains. The structure `myStruct` itself can be defined in a separate translation unit, in this case, consisting of the file `file2.c`. The common header file `file.h` must be included in both `file.c` and `file2.c` for linking the structure definition to the opaque type definition.

`file.h`: Does not contain structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct myStruct *ptrMyStruct;

ptrMyStruct getObj(void);
void useObj(ptrMyStruct);

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

void func() {
  ptrMyStruct sPtr = getObj();
  useObj(sPtr);
}
```

`file2.c`: Includes `file.h` and dereferences structure.

```
#include "file.h"

struct myStruct {
  int a;
};

void useObj(ptrMyStruct ptr) {
    (ptr->a)++;
}
```

### Risk

If a pointer to a structure or union is not dereferenced in a file, the implementation details of the structure or union need not be available in the translation unit for the file. You can hide the implementation details such as structure members and protect them from unintentional changes.

Define an opaque type that can be referenced via pointers but whose contents cannot be accessed.

### Example - Object Implementation Revealed

`file.h`: Contains structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct  {
  int a;
} myStruct;

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

myStruct* getObj(void);
void useObj(myStruct*);

void func() {
  myStruct *sPtr = getObj();
  useObj(sPtr);
}
```

In this example, the pointer to the type `myStruct` is not dereferenced. The pointer is simply obtained from the `getObj` function and passed to the `useObj` function.

The implementation of `myStruct` is visible in the translation unit consisting of `file.c` and `file.h`.

**Correction — Define Opaque Type**

One possible correction is to define an opaque data type in the header file `file.h`. The opaque data type `ptrMyStruct` points to the `myStruct` structure without revealing what the structure contains. The structure `myStruct` itself can be defined in a separate translation unit, in this case, consisting of the file `file2.c`. The common header file `file.h` must be included in both `file.c` and `file2.c` for linking the structure definition to the opaque type definition.

`file.h`: Does not contain structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct myStruct *ptrMyStruct;

ptrMyStruct getObj(void);
void useObj(ptrMyStruct);

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

void func() {
  ptrMyStruct sPtr = getObj();
  useObj(sPtr);
}
```

`file2.c`: Includes `file.h` and dereferences structure.

```
#include "file.h"
```

```
struct myStruct {
  int a;
};

void useObj(ptrMyStruct ptr) {
    (ptr->a)++;
}
```

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL12-C

# CERT C: Rec. DCL13-C

Declare function parameters that are pointers to values not changed by the function as const

## Description

### Rule Definition

*Declare function parameters that are pointers to values not changed by the function as const.*

### Polyspace Implementation

The rule checker checks for **Pointer to non-cont qualified function parameter**.

## Examples

### Pointer to non-cont qualified function parameter

#### Issue

The rule checker flags a pointer to a non-`const` function parameter if the pointer does not modify the addressed object. The assumption is that the pointer is not meant to modify the object and so must point to a `const`-qualified type.

#### Risk

This rule ensures that you do not inadvertently use pointers to modify objects.

#### Example - Pointer That Should Point to const-Qualified Types

```
#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(uint16_t *p) {      /* Non-compliant */
    return *p;
}

char last_char(char * const s){     /* Non-compliant */
    return s[strlen(s) - 1u];
}

uint16_t first(uint16_t a[5]){      /* Non-compliant */
    return a[0];
}
```

This example shows three different noncompliant pointer parameters.

- In the `ptr_ex` function, `p` does not modify an object. However, the type to which `p` points is not `const`-qualified, so it is noncompliant.
- In `last_char`, the pointer `s` is `const`-qualified but the type it points to is not. This parameter is noncompliant because `s` does not modify an object.
- The function `first` does not modify the elements of the array `a`. However, the element type is not `const`-qualified, so `a` is also noncompliant.

#### Correction — Use `const` Keywords

One possible correction is to add `const` qualifiers to the definitions.

```
#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(const uint16_t *p){     /* Compliant */
    return *p;
}

char last_char(const char * const s){   /* Compliant */
    return s[strlen( s ) - 1u];
}

uint16_t first(const uint16_t a[5]) {   /* Compliant */
    return a[0];
}
```

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

DCL13-C

# CERT C: Rec. DCL15-C

Declare file-scope objects or functions that do not need external linkage as static

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*Declare file-scope objects or functions that do not need external linkage as static.*

### Polyspace Implementation

The rule checker checks for **Function or object with external linkage referenced in only one translation unit**.

## Examples

### Function or object with external linkage referenced in only one translation unit

**Issue**

The rule checker flags:

- Objects that are defined at file scope without the `static` specifier but used only in one file.
- Functions that are defined without the `static` specifier but called only in one file.

If you intend to use the object or function in one file only, declare it static.

Objects that are defined at file scope without the `static` specifier but used only in one file.

Functions that are defined without the `static` specifier but called only in one file.

**Risk**

Compliance with this rule avoids confusion between your identifier and an identical identifier in another translation unit or library. If you restrict or reduce the visibility of an object by giving it internal linkage or no linkage, you or someone else is less likely to access the object inadvertently.

**Example - Variable with External Linkage Used in One File**

Header file:

```
/* file.h */
extern int var;
```

First source file:

```
/* file1.c */
#include "file.h"

int var;    /* Compliant */
int var2;   /* Non compliant */
static int var3; /* Compliant */

void reset(void);

void reset(void) {
    var = 0;
    var2 = 0;
    var3 = 0;
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment(int var2);

void increment(int var2) {
    var++;
    var2++;
}
```

In this example:

- The declaration of `var` is compliant because `var` is declared with external linkage and used in multiple files.
- The declaration of `var2` is noncompliant because `var2` is declared with external linkage but used in one file only.

  It might appear that `var2` is defined in both files. However, in the second file, `var2` is a parameter with no linkage and is not the same as the `var2` in the first file.
- The declaration of `var3` is compliant because `var3` is declared with internal linkage (with the `static` specifier) and used in one file only.

#### Example - Function with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
extern void increment1 (void);
```

First source file:

```
/* file1.c */
#include "file.h"

int var;

void increment2(void);
static void increment3(void);
void func(void);

void increment2(void) { /* Non compliant */
    var+=2;
}
```

```
static void increment3(void) { /* Compliant */
    var+=3;
}

void func(void) {
    increment1();
    increment2();
    increment3();
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment1(void) { /* Compliant */
    var++;
}
```

In this example:

- The definition of `increment1` is compliant because `increment1` is defined with external linkage and called in a different file.
- The declaration of `increment2` is noncompliant because `increment2` is defined with external linkage but called in the same file and nowhere else.
- The declaration of `increment3` is compliant because `increment3` is defined with internal linkage (with the `static` specifier) and called in the same file and nowhere else.

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL15-C

# CERT C: Rec. DCL16-C

Use 'L,' not 'l,' to indicate a long value

## Description

### Rule Definition

*Use 'L,' not 'l,' to indicate a long value.*

### Polyspace Implementation

The rule checker checks for **Use of lowercase "l" in literal suffix**.

## Examples

### Use of lowercase "l" in literal suffix

#### Issue

The issue occurs when you use the lowercase character “l” in a literal suffix.

#### Risk

The lowercase character “l” can be confused with the digit “1”. Use the uppercase “L” instead.

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

DCL16-C

# CERT C: Rec. DCL18-C

Do not begin integer constants with 0 when specifying a decimal value

## Description

### Rule Definition

*Do not begin integer constants with 0 when specifying a decimal value.*

### Polyspace Implementation

The rule checker checks for **Use of octal constants**.

## Examples

### Use of octal constants

#### Issue

If you use octal constants in a macro definition, the rule checker flags the issue even if the macro is not used.

#### Risk

Octal constants are denoted by a leading zero. Developers can mistake an octal constant as a decimal constant with a redundant leading zero.

#### Example - Use of octal constants

```
#define CST     021              /* Noncompliant */
#define VALUE   010              /* Noncompliant */
#if 010 == 01                    /* Noncompliant*/
#define CST 021                  /* Noncompliant */
#endif

extern short code[5];
static char* str2 = "abcd\0efg";  /* Compliant */

void main(void) {
    int value1 = 0;               /* Compliant */
    int value2 = 01;              /* Noncompliant*/
    int value3 = 1;               /* Compliant */
    int value4 = '\109';          /* Compliant */

    code[1] = 109;                /* Compliant     - decimal 109 */
    code[2] = 100;                /* Compliant     - decimal 100 */
    code[3] = 052;                /* Noncompliant */
    code[4] = 071;                /* Noncompliant */

    if (value1 != CST) {
        value1 = !(value1 != 0);  /* Compliant */
    }
}
```

In this example, Polyspace flags the use of octal constants.

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL18-C

# CERT C: Rec. DCL19-C

Minimize the scope of variables and functions

## Description

### Rule Definition

*Minimize the scope of variables and functions.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Function or object declared without static specifier and referenced in only one file**.
- **Object defined beyond necessary scope**.

## Examples

### Function or object declared without `static` specifier and referenced in only one file

**Issue**

The rule checker flags:

- Objects that are defined at file scope without the `static` specifier but used only in one file.
- Functions that are defined without the `static` specifier but called only in one file.

If you intend to use the object or function in one file only, declare it static.

Objects that are defined at file scope without the `static` specifier but used only in one file.

Functions that are defined without the `static` specifier but called only in one file.

*The checker does not flag this issue in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.

**Risk**

Compliance with this rule avoids confusion between your identifier and an identical identifier in another translation unit or library. If you restrict or reduce the visibility of an object by giving it internal linkage or no linkage, you or someone else is less likely to access the object inadvertently.

**Example - Variable with External Linkage Used in One File**

Header file:

```
/* file.h */
extern int var;
```

First source file:

```
/* file1.c */
#include "file.h"
```

```
int var;    /* Compliant */
int var2;   /* Non compliant */

void reset(void);

void reset(void) {

    static int var3=0; /* Compliant */
    var = 0;
    var2 = 0;
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment(int var2);

void increment(int var2) {
    var++;
    var2++;
}
```

In this example:

- The declaration of `var` is compliant because `var` is declared with external linkage and used in multiple files.
- The declaration of `var2` is noncompliant because `var2` is declared with external linkage but used in one file only.

  It might appear that `var2` is defined in both files. However, in the second file, `var2` is a parameter with no linkage and is not the same as the `var2` in the first file.
- The declaration of `var3` is compliant because `var3` is declared with internal linkage (with the `static` specifier) and used in one file only.

**Example - Function with External Linkage Used in One File**

Header file:

```
/* file.h */
extern int var;
extern void increment1 (void);
```

First source file:

```
/* file1.c */
#include "file.h"

int var;

void increment2(void);
static void increment3(void);
void func(void);

void increment2(void) { /* Non compliant */
```

```
    var+=2;
}

static void increment3(void) { /* Compliant */
    var+=3;
}

void func(void) {
    increment1();
    increment2();
    increment3();
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment1(void) { /* Compliant */
    var++;
}
```

In this example:

- The definition of `increment1` is compliant because `increment1` is defined with external linkage and called in a different file.
- The declaration of `increment2` is noncompliant because `increment2` is defined with external linkage but called in the same file and nowhere else.
- The declaration of `increment3` is compliant because `increment3` is defined with internal linkage (with the `static` specifier) and called in the same file and nowhere else.

### Object defined beyond necessary scope

#### Issue

The issue occurs when the identifier of an object only appears in a single function but the object is defined beyond the block scope.

The rule checker flags `static` objects that are accessed in one function only but declared at file scope.

#### Risk

If you define an object at block scope, you or someone else is less likely to access the object inadvertently outside the block.

#### Example - Object Declared at File Scope but Used in One Function

```
static int ctr;   /* Non compliant */

int checkStatus(void);
void incrementCount(void);

void incrementCount(void) {
    ctr=0;
    while(1) {
        if(checkStatus())
            ctr++;
    }
}
```

In this example, the declaration of `ctr` is noncompliant because it is declared at file scope but used only in the function `incrementCount`. Declare `ctr` in the body of `incrementCount` to be MISRA C-compliant.

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

DCL19-C

# CERT C: Rec. DCL22-C

Use volatile for data that cannot be cached

## Description

### Rule Definition

*Use volatile for data that cannot be cached.*

### Polyspace Implementation

The rule checker checks for **Write without a further read**.

## Examples

### Write without a further read

#### Issue

**Write without a further read** occurs when a value assigned to a variable is never read.

For instance, you write a value to a variable and then write a second value before reading the previous value. The first write operation is redundant.

#### Risk

Redundant write operations often indicate programming errors. For instance, you forgot to read the variable between two successive write operations or unintentionally read a different variable.

#### Fix

Identify the reason why you write to the variable but do not read it later. Look for common programming errors such as accidentally reading a different variable with a similar name.

If you determine that the write operation is redundant, remove the operation.

#### Example - Write Without Further Read Error

```
void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();             //Noncompliant
    /* Defect: Useless write */
}
```

After the variable `level` gets assigned the value `4 * getsensor()`, it is not read.

#### Correction — Use Value After Assignment

One possible correction is to use the variable `level` after the assignment.

```
#include <stdio.h>

void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();

    /* Fix: Use level after assignment */
    printf("The value is %d", level);

}
```

The variable `level` is printed, reading the new value.

### Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C (-cert-c)`

**Topics**

“Check for and Review Coding Standard Violations”

**External Websites**

DCL22-C

# CERT C: Rec. DCL23-C

Guarantee that mutually visible identifiers are unique

## Description

### Rule Definition

*Guarantee that mutually visible identifiers are unique.*

### Polyspace Implementation

The rule checker checks for these issues:

- **External identifiers not distinct**.
- **Identifier in same scope and namespace not distinct**.
- **Macro identifier not distinct**.
- **Name for macros and identifiers not distinct**.

## Examples

### External identifiers not distinct

#### Issue

The issue occurs when external identifiers have the same first six characters for C90 or the same first 31 characters for C99.

*The checker does not flag this issue in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.

#### Risk

External identifiers are ones declared with global scope or storage class `extern`.

Polyspace considers two names as distinct if there is a difference between their first 31 characters. If the difference between two names occurs only beyond the first 31 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first six characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

#### Example - C90: First Six Characters of Identifiers Not Unique

```
int engine_temperature_raw;
int engine_temperature_scaled;   /* Non-compliant */
int engin2_temperature;          /* Compliant */
```

In this example, the identifier `engine_temperature_scaled` has the same first six characters as a previous identifier, `engine_temperature_raw`.

#### Example - C99: First 31 Characters of Identifiers Not Unique

```
int engine_exhaust_gas_temperature_raw;
int engine_exhaust_gas_temperature_scaled; /* Non-compliant */
int eng_exhaust_gas_temp_raw;
int eng_exhaust_gas_temp_scaled;           /* Compliant */
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same first 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

#### Example - C90: First Six Characters Identifiers in Different Translation Units Differ in Case Alone

```
/* file1.c */
int abc = 0;

/* file2.c */
int ABC = 0; /* Non-compliant */
```

In this example, the implementation supports six significant case-insensitive characters in *external identifiers*. The identifiers in the two translations are different but are not distinct in their significant characters.

### Identifier in same scope and namespace not distinct

#### Issue

The issue occurs when you declare identifiers in the same scope and namespace and the identifiers have the same first 31 characters in C90 or the same first 63 characters in C99.

#### Risk

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

#### Example - C90: First 31 Characters of Identifiers Not Unique

```
extern int engine_exhaust_gas_temperature_raw;
static int engine_exhaust_gas_temperature_scaled;      /* Non-compliant */

extern double engine_exhaust_gas_temperature_raw;
static double engine_exhaust_gas_temperature2_scaled;  /* Compliant */

void func ( void )
{
  /* Not in the same scope */
  int engine_exhaust_gas_temperature_local;            /* Compliant */
}
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

The rule does not apply if the two identifiers have the same 31 characters but have different scopes. For instance, `engine_exhaust_gas_temperature_local` has the same 31 characters as `engine_exhaust_gas_temperature_raw` but different scope.

#### Example - C99: First 63 Characters of Identifiers Not Unique

```
extern int engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_raw;
static int engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_scale;
```

```
    /* Non-compliant */

extern int engine_gas_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx__raw;
static int engine_gas_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx__scale;
    /* Compliant */

void func ( void )
{
/* Not in the same scope */
    int engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_local;
          /* Compliant */
}
```

In this example, the identifier `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_scale` has the same 63 characters as a previous identifier, `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_raw`.

### Macro identifier not distinct

#### Issue

The issue occurs when you use macro identifiers that have the same first 31 characters in C90 or the same first 63 characters in C99.

#### Risk

The names of macro identifiers must be distinct from both other macro identifiers and their parameters.

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

#### Example - C90: First 31 Characters of Macro Names Not Unique

```
#define engine_exhaust_gas_temperature_raw egt_r
#define engine_exhaust_gas_temperature_scaled egt_s   /* Non-compliant */

#define engine_exhaust_gas_temp_raw egt_r
#define engine_exhaust_gas_temp_scaled egt_s          /* Compliant */
```

In this example, the macro `engine_exhaust_gas_temperature_scaled egt_s` has the same first 31 characters as a previous macro `engine_exhaust_gas_temperature_scaled`.

#### Example - C99: First 63 Characters of Macro Names Not Unique

```
#define engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_raw egt_r
#define engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_raw_scaled egt_s
    /* Non-compliant */

/* 63 significant case-sensitive characters in macro identifiers */
#define new_engine_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_raw egt_r
#define new_engine_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_scaled egt_s
    /* Compliant */
```

In this example, the macro `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx___gaz_scaled` has the same first 63 characters as a previous macro `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx___raw`.

#### Name for macros and identifiers not distinct

##### Issue

The issue occurs when identifiers are not distinct from macro names.

##### Risk

The rule requires that macro names that exist only prior to processing must be different from identifier names that also exist after preprocessing. Keeping macro names and identifiers distinct help avoid confusion.

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

##### Example - Macro Names Same as Identifier Names

```
#define Sum_1(x, y) ( ( x ) + ( y ) )
short Sum_1;                    /* Non-compliant */

#define Sum_2(x, y) ( ( x ) + ( y ) )
short x = Sum_2 ( 1, 2 );       /* Compliant */
```

In this example, `Sum_1` is both the name of an identifier and a macro. `Sum_2` is used only as a macro.

##### Example - C90: First 31 Characters of Macro Name Same as Identifier Name

```
#define       low_pressure_turbine_temperature_1 lp_tb_temp_1
static int low_pressure_turbine_temperature_2;      /* Non-compliant  */
```

In this example, the identifier `low_pressure_turbine_temperature_2` has the same first 31 characters as a previous macro `low_pressure_turbine_temperature_1`.

## Check Information

**Group:** Rec. 02. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

DCL23-C

# CERT C: Rec. EXP00-C

Use parentheses for precedence of operation

## Description

### Rule Definition

*Use parentheses for precedence of operation.*

### Polyspace Implementation

The rule checker checks for **Possibly unintended evaluation of expression because of operator precedence rules**.

## Examples

### Possibly unintended evaluation of expression because of operator precedence rules

#### Issue

**Possibly unintended evaluation of expression because of operator precedence rules** occurs when an arithmetic expression result is possibly unintended because operator precedence rules dictate an evaluation order that you do not expect.

The defect highlights expressions of the form `x op_1 y op_2 z`. Here, *op_1* and *op_2* are operator combinations that commonly induce this error. For instance, `x == y | z`.

The checker does not flag all operator combinations. For instance, `x == y || z` is not flagged because you most likely intended to perform a logical OR between `x == y` and `z`. Specifically, the checker flags these combinations:

- && and ||: For instance, `x || y && z` or `x && y || z`.
- Assignment and bitwise operations: For instance, `x = y | z`.
- Assignment and comparison operations: For instance, `x = y != z` or `x = y > z`.
- Comparison operations: For instance, `x > y > z` (except when one of the comparisons is an equality `x == y > z`).
- Shift and numerical operation: For instance, `x << y + 2`.
- Pointer dereference and arithmetic: For instance, `*p++`.

#### Risk

The defect can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:
  - In the operation `*p++`, it is possible that you expect the dereferenced value to be incremented. However, the pointer `p` is incremented before the dereference.

- In the operation `(x == y | z)`, it is possible that you expect `x` to be compared with `y | z`. However, the == operation happens before the | operation.

**Fix**

See if the order of evaluation is what you intend. If not, apply parentheses to implement the evaluation order that you want.

For better readability of your code, it is good practice to apply parenthesis to implement an evaluation order even when operator precedence rules impose that order.

**Example - Expressions with Possibly Unintended Evaluation Order**

```
int test(int a, int b, int c) {
    return(a & b == c); //Noncompliant
}
```

In this example, the == operation happens first, followed by the & operation. If you intended the reverse order of operations, the result is not what you expect.

**Correction — Parenthesis For Intended Order**

One possible correction is to apply parenthesis to implement the intended evaluation order.

```
int test(int a, int b, int c) {
    return((a & b) == c);
}
```

## Check Information

**Group:** Rec. 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**

“Check for and Review Coding Standard Violations”

**External Websites**

EXP00-C

# CERT C: Rec. EXP05-C

Do not cast away a const qualification

## Description

### Rule Definition

*Do not cast away a const qualification.*

### Polyspace Implementation

The rule checker checks for **Cast to pointer that removes const qualification**.

## Examples

### Cast to pointer that removes `const` qualification

#### Issue

Polyspace flags both implicit and explicit conversions that violate this rule.

#### Risk

This rule forbids casts from a pointer to a `const` object to a pointer that does not point to a `const` object.

Such casts violate type qualification. For example, the `const` qualifier indicates the read-only status of an object. If a cast removes the qualifier, the object is no longer read-only.

#### Example - Casts That Remove Qualifiers

```
void foo(void) {

    /* Cast on simple type */
    unsigned short           x;
    unsigned short * const   cpi = &x;  /* const pointer */
    unsigned short * const  *pcpi;   /* pointer to const pointer */
    unsigned short **ppi;
    const unsigned short    *pci;    /* pointer to const */
    unsigned short          *pi;

    pi = cpi;                        /* Compliant - no cast required */
    pi  = (unsigned short *)  pci;   /* Non-compliant */
    ppi = (unsigned short **)pcpi;   /* Non-compliant */
}
```

In this example, the variables `pci` and `pcpi` have the `const` qualifier in their type. The rule is violated when the variables are cast to types that do not have the `const` qualifier.

Even though `cpi` has a `const` qualifier in its type, the rule is not violated in the statement `p=cpi;`. The assignment does not cause a type conversion because both `p` and `cpi` have type `unsigned short`.

## Check Information

**Group:** Rec. 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP05-C

# CERT C: Rec. EXP08-C

Ensure pointer arithmetic is used correctly

## Description

### Rule Definition

*Ensure pointer arithmetic is used correctly.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access out of bounds**
- **Pointer access out of bounds**.
- **Subtraction between pointers to different arrays**.
- **Incorrect pointer scaling**.

### Extend Checker

A default Bug Finder analysis might not flag an **Array access out of bounds** issue when the input values are unknown and only a subset of inputs cause the issue. To check for the **Array access out of bounds** issue caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Array access out of bounds

#### Issue

This issue occurs when an array index falls outside the range `[0...array_size-1]` during array access.

#### Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example – Array Access Out of Bounds Error**

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);    //Noncompliant
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

**Correction – Keep Array Index Within Array Bounds**

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];
```

```
    for (i = 0; i < 10; i++)
     {
         if (i < 2)
             fib[i] = 1;
         else
             fib[i] = fib[i-1] + fib[i-2];
     }

     /* Fix: Print fib[9] instead of fib[10] */
     printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

### Pointer access out of bounds

#### Issue

This issue occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

#### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example – Pointer access out of bounds error**

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i; //Noncompliant
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

**Correction — Check Pointer Stays Within Bounds**

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

### Subtraction between pointers to different arrays

**Issue**

This rule is raised whenever the analysis detects a `Subtraction or comparison between pointers to different arrays`.

**Risk**

This rule applies to expressions of the form `pointer_expression1 - pointer_expression2`. The behavior is undefined if `pointer_expression1` and `pointer_expression2`:

- Do not point to elements of the same array,
- Or do not point to the element one beyond the end of the array.

**Example - Subtracting Pointers**

```
#include <stdint.h>
#include <stddef.h>

void f1 (int32_t *ptr)
{
    int32_t a1[10];
    int32_t a2[10];
    int32_t *p1 = &a1[ 1];
    int32_t *p2 = &a2[10];
    ptrdiff_t diff1, diff2, diff3;

    diff1 =  p1 - a1;   // Compliant
    diff2 =  p2 - a2;   // Compliant
    diff3 =  p1 - p2;   // Non-compliant
}
```

In this example, the three subtraction expressions show the difference between compliant and noncompliant pointer subtractions. The `diff1` and `diff2` subtractions are compliant because the pointers point to the same array. The `diff3` subtraction is not compliant because `p1` and `p2` point to different arrays.

## Incorrect pointer scaling

### Issue

**Incorrect pointer scaling** occurs when Polyspace Bug Finder considers that you are ignoring the implicit scaling in pointer arithmetic.

For instance, the defect can occur in the following situations.

| Situation | Risk | Possible Fix |
|---|---|---|
| You use the `sizeof` operator in arithmetic operations on a pointer. | The `sizeof` operator returns the size of a data type in number of bytes.<br>Pointer arithmetic is already implicitly scaled by the size of the data type of the pointed variable. Therefore, the use of `sizeof` in pointer arithmetic produces unintended results. | Do not use `sizeof` operator in pointer arithmetic. |
| You perform arithmetic operations on a pointer, and then apply a cast. | Pointer arithmetic is implicitly scaled. If you do not consider this implicit scaling, casting the result of a pointer arithmetic produces unintended results. | Apply the cast before the pointer arithmetic. |

### Fix

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Example - Use of sizeof Operator

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2*(sizeof(int))); //Noncompliant
}
```

In this example, the operation `2*(sizeof(int))` returns twice the size of an `int` variable in bytes. However, because pointer arithmetic is implicitly scaled, the number of bytes by which `ptr` is offset is `2*(sizeof(int))*(sizeof(int))`.

In this example, the incorrect scaling shifts `ptr` outside the bounds of the array. Therefore, a **Pointer access out of bounds** error appears on the * operation.

### Correction — Remove sizeof Operator

One possible correction is to remove the `sizeof` operator.

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2);
}
```

### Example - Cast Following Pointer Arithmetic

```
int func(void) {
    int x = 0;
    char r = *(char *)(&x + 1); //Noncompliant
    return r;
}
```

In this example, the operation `&x + 1` offsets `&x` by `sizeof(int)`. Following the operation, the resulting pointer points outside the allowed buffer. When you dereference the pointer, a **Pointer access out of bounds** error appears on the * operation.

### Correction — Apply Cast Before Pointer Arithmetic

If you want to access the second byte of `x`, first cast `&x` to a `char*` pointer and then perform the pointer arithmetic. The resulting pointer is offset by `sizeof(char)` bytes and still points within the allowed buffer, whose size is `sizeof(int)` bytes.

```
int func(void) {
    int x = 0;
    char r = *((char *)(&x )+ 1);
    return r;
}
```

## Check Information

**Group:** Rec. 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP08-C

# CERT C: Rec. EXP09-C

Use sizeof to determine the size of a type or variable

## Description

### Rule Definition

*Use sizeof to determine the size of a type or variable.*

### Polyspace Implementation

The rule checker checks for **Hard-coded object size used to manipulate memory**.

## Examples

### Hard-coded object size used to manipulate memory

#### Issue

The issue **Hard-coded object size used to manipulate memory** occurs when you use hard-coded constants as memory size arguments for these memory functions:

- Dynamic memory allocation function such as `malloc` or `calloc`.
- Memory manipulation functions such as `memcpy`, `memmove`, `memcmp`, or `memset`.

When performing memory operations with a string literal, Polyspace does not report a violation of this rule if you hard code the memory size.

#### Risk

If you hard code object size, your code is not portable to architectures with different type sizes. If the constant value is not the same as the object size, the buffer might or might not overflow.

#### Fix

For the size argument of memory functions, use `sizeof(`*`object`*`)`.

#### Example - Assume 4-Byte Integer Pointers

```
#include <stddef.h>
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};
extern void fill_ints(int **matrix, size_t nb, size_t s);

void bug_hardcodedmemsize()
{
    size_t i, s;

    s = 4;
    int **matrix = (int **)calloc(SIZE20, s); //Noncompliant
    if (matrix == NULL) {
        return; /* Indicate calloc() failure */
    }
    fill_ints(matrix, SIZE20, s);
    free(matrix);
}
```

In this example, the memory allocation function `calloc` is called with a memory size of 4. The memory is allocated for an integer pointer, which can be a more or less than 4 bytes depending on your target. If the integer pointer is not 4 bytes, your program can fail.

**Correction — Use `sizeof(int *)`**

When calling `calloc`, replace the hard-coded size with a call to `sizeof`. This change makes your code more portable.

```
#include <stddef.h>
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};
extern void fill_ints(int **matrix, size_t nb, size_t s);

void corrected_hardcodedmemsize()
{
    size_t i, s;

    s = sizeof(int *);
    int **matrix = (int **)calloc(SIZE20, s);
    if (matrix == NULL) {
        return; /* Indicate calloc() failure */
    }
    fill_ints(matrix, SIZE20, s);
    free(matrix);
}
```

**Hard Coded Size in `memset`**

In this example, the function `clean_sensitive_memory` clears sensitive information from the memory. Here, the memory size argument of `memset` is hardcoded to be 64 bytes. If `s->data` cannot accommodate `64` bytes, the program fails and the sensitive information might remain in memory. Polyspace reports a violation of this rule on the memory operations.

```
#include<string.h>

struct sensitiveInfo
{
    unsigned char data[64];
    int length;
};

char key[64];

void clean_sensitive_memory (struct sensitiveInfo *s)
{
    memset (s->data, 0, 64);          //Defect
    memset ((void *) key, 0, 64);  //Defect
}
```

#### Correction — Use `sizeof()` for Size Argument

To fix this violation, replace the hardcoded memory sizes by calls to `sizeof()`.

```
#include<string.h>

struct sensitiveInfo
{
    unsigned char data[64];
    int length;
};

char key[64];

void clean_sensitive_memory (struct sensitiveInfo *s)
{
  memset (s->data, 0, sizeof (s->data));    //Fixed
  memset ((void *) key, 0, sizeof(key)); //Fixed
}
```

## Check Information

**Group:** Rec. 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP09-C

# CERT C: Rec. EXP10-C

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

## Description

### Rule Definition

*Do not depend on the order of evaluation of subexpressions or the order in which side effects take place.*

### Polyspace Implementation

The rule checker checks for **Expression value depends on order of evaluation or of side effects**.

## Examples

### Expression value depends on order of evaluation or of side effects

#### Issue

The issue occurs when the value of an expression and its persistent side effects is not the same under all permitted evaluation orders.

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.

Therefore, this rule forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation.

#### Risk

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

#### Example - Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         /* Compliant */
    COPY_ELEMENT (i++);       /* Noncompliant  */
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

**Example - Variable Modified and Used in Multiple Function Arguments**

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
    f ( i++, i );                 /* Non-compliant */
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

## Check Information

**Group:** Rec. 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

EXP10-C

# CERT C: Rec. EXP12-C

Do not ignore values returned by functions

## Description

### Rule Definition

*Do not ignore values returned by functions.*

### Polyspace Implementation

The rule checker checks for **Returned value of a sensitive function not checked**.

## Examples

### Returned value of a sensitive function not checked

#### Issue

**Returned value of a sensitive function not checked** occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

#### Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

#### Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

#### Example - Sensitive Function Return Ignored

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);  //Noncompliant
}
```

This example shows a call to the sensitive function `pthread_attr_init`. The return value of `pthread_attr_init` is ignored, causing a defect.

#### Correction — Cast Function to (void)

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);
}
```

#### Correction — Test Return Value

One possible correction is to test the return value of `pthread_attr_init` to check for errors.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

void initialize() {
    pthread_attr_t attr;
    int result;

    result = pthread_attr_init(&attr);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

#### Example - Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0)); //Noncompliant
    pthread_join(thread_id,  &res);  //Noncompliant
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

#### Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

## Check Information

**Group:** Rec. 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP12-C

# CERT C: Rec. EXP13-C

Treat relational and equality operators as if they were nonassociative

## Description

### Rule Definition

*Treat relational and equality operators as if they were nonassociative.*

### Polyspace Implementation

The rule checker checks for **Possibly unintended evaluation of expression because of operator precedence rules**.

## Examples

### Possibly unintended evaluation of expression because of operator precedence rules

#### Issue

**Possibly unintended evaluation of expression because of operator precedence rules** occurs when an arithmetic expression result is possibly unintended because operator precedence rules dictate an evaluation order that you do not expect.

The defect highlights expressions of the form `x` *op_1* `y` *op_2* `z`. Here, *op_1* and *op_2* are operator combinations that commonly induce this error. For instance, `x == y | z`.

The checker does not flag all operator combinations. For instance, `x == y || z` is not flagged because you most likely intended to perform a logical OR between `x == y` and `z`. Specifically, the checker flags these combinations:

- `&&` and `||`: For instance, `x || y && z` or `x && y || z`.
- Assignment and bitwise operations: For instance, `x = y | z`.
- Assignment and comparison operations: For instance, `x = y != z` or `x = y > z`.
- Comparison operations: For instance, `x > y > z` (except when one of the comparisons is an equality `x == y > z`).
- Shift and numerical operation: For instance, `x << y + 2`.
- Pointer dereference and arithmetic: For instance, `*p++`.

#### Risk

The defect can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:
  - In the operation `*p++`, it is possible that you expect the dereferenced value to be incremented. However, the pointer `p` is incremented before the dereference.

- In the operation `(x == y | z)`, it is possible that you expect `x` to be compared with `y | z`. However, the == operation happens before the | operation.

**Fix**

See if the order of evaluation is what you intend. If not, apply parentheses to implement the evaluation order that you want.

For better readability of your code, it is good practice to apply parenthesis to implement an evaluation order even when operator precedence rules impose that order.

**Example - Expressions with Possibly Unintended Evaluation Order**

```
int test(int a, int b, int c) {
    return(a & b == c); //Noncompliant
}
```

In this example, the == operation happens first, followed by the & operation. If you intended the reverse order of operations, the result is not what you expect.

**Correction — Parenthesis For Intended Order**

One possible correction is to apply parenthesis to implement the intended evaluation order.

```
int test(int a, int b, int c) {
    return((a & b) == c);
}
```

## Check Information

**Group:** Rec. 03. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

EXP13-C

# CERT C: Rec. EXP15-C

Do not place a semicolon on the same line as an if, for, or while statement

## Description

### Rule Definition

*Do not place a semicolon on the same line as an if, for, or while statement.*

### Polyspace Implementation

The rule checker checks for **Semicolon on same line as if, for or while statement**.

## Examples

### Semicolon on same line as if, for or while statement

#### Issue

**Semicolon on same line as if, for or while statement** occurs when a semicolon on the same line as the last token of an `if`, `for` or `while` statement results in an empty body.

The checker makes an exception for the case where the `if` statement is immediately followed by an `else` statement:

```
if(condition);
else {
  //...
}
```

#### Risk

The semicolon following the `if`, `for` or `while` statement often indicates a programming error. The spurious semicolon changes the execution flow and leads to unintended results.

#### Fix

If you want an empty body for the `if`, `for` or `while` statement , wrap the semicolon in a block and place the block on a new line to explicitly indicate your intent:

```
if(condition)
   {;}
```

Otherwise, remove the spurious semicolon.

#### Example - Spurious Semicolon

```
int credentialsOK(void);

void login () {
    int loggedIn = 0;
    if(credentialsOK()); //Noncompliant
      loggedIn = 1;
}
```

In this example, the spurious semicolon results in an empty `if` body. The assignment `loggedIn=1` is always performed. However, the assignment was probably to be performed only under a condition.

#### Correction – Remove Spurious Semicolon

If the semicolon was unintended, remove the semicolon.

```
int credentialsOK(void);

void login () {
    int loggedIn = 0;
    if(credentialsOK())
      loggedIn = 1;
}
```

## Check Information

**Group:** Rec. 03. Expressions (EXP)

# Version History

**Introduced in R2020a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP15-C

# CERT C: Rec. EXP19-C

Use braces for the body of an if, for, or while statement

## Description

### Rule Definition

*Use braces for the body of an if, for, or while statement.*

### Polyspace Implementation

The rule checker checks for **Iteration or selection statement body not enclosed in braces**.

## Examples

### Iteration or selection statement body not enclosed in braces

#### Issue

The issue occurs when you do not enclose the body of an iteration-statement or a selection-statement in braces.

#### Risk

The rule applies to:

- Iteration statements such as `while`, `do ... while` or `for`.
- Selection statements such as `if ... else` or `switch`.

If the block of code associated with an iteration or selection statement is not contained in braces, you can make mistakes about the association. For example:

- You can wrongly associate a line of code with an iteration or selection statement because of its indentation.
- You can accidentally place a semicolon following the iteration or selection statement. Because of the semicolon, the line following the statement is no longer associated with the statement even though you intended otherwise.

#### Example - Iteration Block

```
int data_available = 1;
void f1(void) {
    while(data_available)                /* Non-compliant */
        process_data();

    while(data_available) {              /* Compliant */
        process_data();
    }
}
```

In this example, the second `while` block is enclosed in braces and does not violate the rule.

**Example - Nested Selection Statements**

```
#include <stdbool.h>

bool flag_1, flag_2;

void f1(void) {
    if(flag_1)                          /* Non-compliant */
        if(flag_2)                      /* Non-compliant */
            action_1();
    else                                /* Non-compliant */
            action_2();
}
```

In this example, the rule is violated because the `if` or `else` blocks are not enclosed in braces. Unless indented as above, it is easy to associate the `else` statement with the inner `if`.

**Correction — Place Selection Statement Block in Braces**

One possible correction is to enclose each block associated with an `if` or `else` statement in braces.

```
#include <stdbool.h>

bool flag_1, flag_2;

void f1(void) {
    if(flag_1) {                        /* Compliant */
        if(flag_2) {                      /* Compliant */
            action_1();
        }
    }
    else {                              /* Compliant */
        action_2();
    }
}
```

**Example - Spurious Semicolon After Iteration Statement**

```
#include <stdbool.h>

bool flag_1;

void f1(void) {
    while(flag_1);                      /* Non-compliant */
    {
        flag_1 = action_1();
    }
}
```

In this example, the rule is violated even though the `while` statement is followed by a block in braces. The semicolon following the `while` statement causes the block to dissociated from the `while` statement.

The rule helps detect such spurious semicolons.

## Check Information

**Group:** Rec. 03. Expressions (EXP)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP19-C

# CERT C: Rec. INT00-C

Understand the data model used by your implementation(s)

## Description

### Rule Definition

*Understand the data model used by your implementation(s).*

### Polyspace Implementation

The rule checker checks for these issues:

- **Use of basic types declarations and definitions of variables or functions**.
- **Integer overflow**.
- **Integer constant overflow**.
- **Format string specifiers and arguments mismatch**.

### Extend Checker

A default Bug Finder analysis might not detect **Integer constant overflow** and **Integer constant overflow** when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

## Examples

### Use of basic types declarations and definitions of variables or functions

#### Issue

The issue occurs when you use basic numerical types instead of `typedefs` that indicate size and signedness.

The rule checker flags use of basic data types in variable or function declarations and definitions. The rule enforces use of `typedefs` instead.

The rule checker does not flag the use of basic types in the `typedef` statements themselves.

#### Risk

When the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object.

#### Example - Direct Use of Basic Types in Definitions

```
typedef unsigned int uint32_t;

int x = 0;       /* Non compliant */
uint32_t y = 0;  /* Compliant */
```

In this example, the declaration of `x` is noncompliant because it uses a basic type directly.

### Integer overflow

#### Issue

**Integer overflow** occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer overflows on signed integers result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Addition of Maximum Integer

```
#include <limits.h>
typedef int int32;
int32 plusplus(void) {

    int32 var = INT_MAX;
    var++;   //Noncompliant           //Noncompliant
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

#### Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>
typedef int int32;
typedef long long llint;
llint plusplus(void) {

    llint var = INT_MAX;
    var++;   //Compliant
    return var;
}
```

### Integer constant overflow

#### Issue

**Integer constant overflow** occurs when you assign a compile-time constant to a signed integer variable whose data type cannot accommodate the value. An `n`-bit signed integer holds values in the range $[-2^{n-1}, 2^{n-1}-1]$.

For instance, `c` is an 8-bit signed `char` variable that cannot hold the value 255.

```
signed char c = 255;
```

To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.

#### Risk

The default behavior for constant overflows can vary between compilers and platforms. Retaining constant overflows can reduce the portability of your code.

Even if your compilers wraps around overflowing constants with a warning, the wrap-around behavior can be unintended and cause unexpected results.

#### Fix

Check if the constant value is what you intended. If the value is correct, use a different, possibly wider, data type for the variable.

#### Example - Overflowing Constant from Macro Expansion

```
#define MAX_UNSIGNED_CHAR 255  //Noncompliant
#define MAX_SIGNED_CHAR 127 //Noncompliant

void main() {
    char c1 = MAX_UNSIGNED_CHAR;
    char c2 = MAX_SIGNED_CHAR+1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow. To reproduce these defects, use a `Target processor type (-target)` where `char` is signed by default.

#### Correction — Use Different Data Type

One possible correction is to use a different data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127
typedef unsigned char uchar;
void main() {
    uchar c1 = MAX_UNSIGNED_CHAR;
    uchar c2 = MAX_SIGNED_CHAR+1;
}
```

### Format string specifiers and arguments mismatch

#### Issue

**Format string specifiers and arguments mismatch** occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

#### Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

#### Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
  const char *message = "License not available";
  int err_number = ;-4
  printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Printing a Float

```
#include <stdio.h>
typedef unsigned long UL;
void string_format(void) {

    UL fst = 1;
    printf("%d\n", fst); //Noncompliant //Noncompliant
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

#### Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>
typedef unsigned long UL;
typedef int int32;
void string_format(void) {

    UL fst = 1;

    printf("%lu\n", fst); //Compliant
}
```

#### Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>
typedef unsigned long UL;
typedef int int32;
void string_format(void) {

    UL fst = 1;

    printf("%d\n", (int32)fst); //Compliant
}
```

## Check Information

**Group:** Rec. 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

INT00-C

# CERT C: Rec. INT02-C

Understand integer conversion rules

## Description

### Rule Definition

*Understand integer conversion rules.*

### Polyspace Implementation

The rule checker checks for **Sign change integer conversion overflow**.

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Sign change integer conversion overflow

#### Issue

**Sign change integer conversion overflow** occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Convert from `unsigned char` to `char`

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count; //Noncompliant
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

#### Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

## Check Information

**Group:** Rec. 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

INT02-C

# CERT C: Rec. INT04-C

Enforce limits on integer values originating from tainted sources

## Description

### Rule Definition

*Enforce limits on integer values originating from tainted sources.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access with tainted index**.
- **Loop bounded with tainted value**.
- **Memory allocation with tainted size**.
- **Tainted size of variable length array**.

### Extend Checker

A default Bug Finder analysis might not flag this checker for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Array access with tainted index

#### Issue

**Array access with tainted index** detects reading or writing to an array by using a tainted index that has not been validated.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

#### Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

#### Example - Use Index to Return Buffer Value

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];   //Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The index `num` is obtained from an unsecure source and the function `taintedarrayindex` does not check to see if `num` is inside the range of `tab`.

#### Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

### Loop bounded with tainted value

#### Issue

**Loop bounded with tainted value** detects loops that are bounded by values from an unsecure source.

#### Risk

A tainted value can cause over looping or infinite loops. Attackers can use this vulnerability to crash your program or cause other unintended behavior.

#### Fix

Before starting the loop, validate unknown boundary and iterator values.

#### Example - Loop Boundary From Input Argument

```
#include<stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedloopboundary(void) {
    int count;
    scanf("%d", &count);
    int res = 0;
    for (int i=0 ; i < count; ++i) { //Noncompliant
        res += i;
    }
    return res;
}
```

In this example, the function uses the input argument to loop `count` times. `count` could be any number because the value is not checked before starting the `for` loop.

#### Correction: Clamp Tainted Loop Control

One possible correction is to clamp the tainted loop control. To validate the tainted loop variable `count`, this example limits `count` to a minimum value and a maximum value by using inline functions `min` and `max`. Regardless of the user input, the value of `count` remains within a known range.

```
#include<stdio.h>
#include<algorithm>
#define MIN 50
#define MAX 128
static  inline int max(int a, int b) { return a > b ? a : b;}
static inline int min(int a, int b) { return a < b ? a : b; }

int taintedloopboundary(void) {
    int count;
    scanf("%d", &count);
    int res = 0;
    count = max(MIN, min(count, MAX));
    for (int i=0 ; i<count ; ++i) {
        res += i;
    }
    return res;
}
```

#### Correction — Check Tainted Loop Control

Another possible correction is to check the low bound and the high bound of the tainted loop boundary variable before starting the `for` loop. This example checks the low and high bounds of `count` and executes the loop only when `count` is between 0 and 127.

```
#include<stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};


int taintedloopboundary(void) {
    int count;
    scanf("%d", &count);
    int res = 0;

    if (count>=0 && count<SIZE128) {
        for (int i=0 ; i<count ; ++i) {
            res += i;
        }
    }
    return res;
}
```

### Memory allocation with tainted size

#### Issue

**Memory allocation with tainted size** checks memory allocation functions, such as `calloc` or `malloc`, for size arguments from unsecured sources.

#### Risk

Uncontrolled memory allocation can cause your program to request too much system memory. This consequence can lead to a crash due to an out-of-memory condition, or assigning too many resources.

#### Fix

Before allocating memory, check the value of your arguments to check that they do not exceed the bounds.

#### Example - Allocate Memory Using Input From User

```
#include<stdio.h>
#include <stdlib.h>

int* bug_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = (int*)malloc(size); //Noncompliant
    return p;
}
```

In this example, `malloc` allocates `size` bytes of memory for the pointer `p`. The variable `size` comes from the user of the program. Its value is not checked, and it could be larger than the amount of available memory. If `size` is larger than the number of available bytes, your program could crash.

#### Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the `malloc` operation. This example checks to see if `size` is positive and less than the maximum size.

```
#include<stdio.h>
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int* corrected_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = NULL;
    if (size>0 && size<SIZE128) {          /* Fix: Check entry range before use */
        p = (int*)malloc((unsigned int)size);
    }
    return p;
}
```

### Tainted size of variable length array

#### Issue

**Tainted size of variable length array** detects variable length arrays (VLA) whose size is from an unsecure source.

#### Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

#### Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

#### Example - Input Argument Used as Size of VLA

```
#include<stdio.h>
#inclule<stdlib.h>
#define LIM 40

long squaredSum(int size) {

    int tabvla[size]; //Noncompliant
    long res = 0;
    for (int i=0 ; i<LIM-1 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

#### Correction — Check VLA Size

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 0 and less than 40, before creating the VLA

```
#include <stdio.h>
#include <stdlib.h>
#define LIM 40

long squaredSum(int size) {
    long res = 0;
    if (size>0 && size<LIM){
        int tabvla[size];
        for (int i=0 ; i<size || i<LIM-1 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }else{
        res = -1;
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

## Check Information

**Group:** Rec. 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C (-cert-c)

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

INT04-C

# CERT C: Rec. INT07-C

Use only explicitly signed or unsigned char type for numeric values

## Description

### Rule Definition

*Use only explicitly signed or unsigned char type for numeric values.*

### Polyspace Implementation

The rule checker checks for **Use of plain char type for numerical value**.

## Examples

### Use of plain char type for numerical value

#### Issue

**Use of plain char type for numerical value** detects `char` variables without explicit signedness that are being used in these ways:

- To store non-char constants
- In an arithmetic operation when the `char` is:
  - A negative value.
  - The result of a sign changing overflow.
- As a buffer offset.

`char` variables without a `signed` or `unsigned` qualifier can be either signed or unsigned depending on your compiler.

#### Risk

Operations on a plain char can result in unexpected numerical values. If the char is used as an offset, the char can cause buffer overflow or underflow.

#### Fix

When initializing a char variable, to avoid implementation-defined confusion, explicitly state whether the char is signed or unsigned.

#### Example - Divide by `char` Variable

```
#include <stdio.h>

void badplaincharuse(void)
{
    char c = 200;
    int i = 1000;
    (void)printf("i/c = %d\n", i/c); //Noncompliant
}
```

In this example, the char variable `c` can be signed or unsigned depending on your compiler. Assuming 8-bit, two's complement character types, the result is either `i/c = 5` (unsigned char) or `i/c = -17` (signed char). The correct result is unknown without knowing the signedness of `char`.

#### Correction — Add `signed` Qualifier

One possible correction is to add a `signed` qualifier to `char`. This clarification makes the operation defined.

```
#include <stdio.h>

void badplaincharuse(void)
{
    signed char c = -56;
    int i = 1000;
    (void)printf("i/c = %d\n", i/c);
}
```

## Check Information

**Group:** Rec. 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

INT07-C

# CERT C: Rec. INT08-C

Verify that all integer values are in range

## Description

### Rule Definition

*Verify that all integer values are in range.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Integer overflow**.
- **Integer constant overflow**.

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Integer overflow

#### Issue

**Integer overflow** occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer overflows on signed integers result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.

- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;             //Noncompliant
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

#### Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

### Integer constant overflow

#### Issue

**Integer constant overflow** occurs when you assign a compile-time constant to a signed integer variable whose data type cannot accommodate the value. An `n`-bit signed integer holds values in the range $[-2^{n-1}, 2^{n-1}-1]$.

For instance, `c` is an 8-bit signed `char` variable that cannot hold the value 255.

```
signed char c = 255;
```

To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.

#### Risk

The default behavior for constant overflows can vary between compilers and platforms. Retaining constant overflows can reduce the portability of your code.

Even if your compilers wraps around overflowing constants with a warning, the wrap-around behavior can be unintended and cause unexpected results.

#### Fix

Check if the constant value is what you intended. If the value is correct, use a different, possibly wider, data type for the variable.

#### Example - Overflowing Constant from Macro Expansion

```
#define MAX_UNSIGNED_CHAR 255  //Noncompliant
#define MAX_SIGNED_CHAR 127 //Noncompliant

void main() {
    char c1 = MAX_UNSIGNED_CHAR;
    char c2 = MAX_SIGNED_CHAR+1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow. To reproduce these defects, use a `Target processor type (-target)` where `char` is signed by default.

#### Correction — Use Different Data Type

One possible correction is to use a different data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR;
    unsigned char c2 = MAX_SIGNED_CHAR+1;
}
```

## Check Information

**Group:** Rec. 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**
INT08-C

# CERT C: Rec. INT09-C

Ensure enumeration constants map to unique values

## Description

### Rule Definition

*Ensure enumeration constants map to unique values.*

### Polyspace Implementation

The rule checker checks for **Enumeration constants map to same value**.

## Examples

### Enumeration constants map to same value

#### Issue

The issue occurs when, within an enumerator list, the value of an implicitly-specified enumeration constants are not unique.

The rule checker flags an enumeration if it has an implicitly specified enumeration constant with the same value as another enumeration constant.

#### Risk

An implicitly specified enumeration constant has a value one greater than its predecessor. If the first enumeration constant is implicitly specified, then its value is 0. An explicitly specified enumeration constant has the specified value.

If implicitly and explicitly specified constants are mixed within an enumeration list, it is possible for your program to replicate values. Such replications can be unintentional and can cause unexpected behavior.

#### Example - Replication of Value in Implicitly Specified Enum Constants

```
enum color1 {red_1, blue_1, green_1};   /* Compliant */
enum color2 {red_2 = 1, blue_2 = 2, green_2 = 3};       /* Compliant */
enum color3 {red_3 = 1, blue_3, green_3};     /* Compliant */
enum color4 {red_4, blue_4, green_4 = 1};     /* Non Compliant */
enum color5 {red_5 = 2, blue_5, green_5 = 2};     /* Compliant */
enum color6 {red_6 = 2, blue_6, green_6 = 2, yellow_6};     /* Non Compliant */
```

Compliant situations:

- `color1`: All constants are implicitly specified.
- `color2`: All constants are explicitly specified.
- `color3`: Though there is a mix of implicit and explicit specification, all constants have unique values.
- `color5`: The implicitly specified constants have unique values.

Noncompliant situations:

- `color4`: The implicitly specified constant `blue_4` has the same value as `green_4`.
- `color6`: The implicitly specified constant `blue_6` has the same value as `yellow_6`.

## Check Information

**Group:** Rec. 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

INT09-C

# CERT C: Rec. INT10-C

Do not assume a positive remainder when using the % operator

## Description

### Rule Definition

*Do not assume a positive remainder when using the % operator.*

### Polyspace Implementation

The rule checker checks for **Tainted modulo operand**.

### Extend Checker

A default Bug Finder analysis might not flag a **Tainted modulo operand** issue for certain inputs that originate outside of the current analysis boundary. See "Sources of Tainting in a Polyspace Analysis". To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Tainted modulo operand

#### Issue

**Tainted modulo operand** checks the operands of remainder `%` operations. Bug Finder flags modulo operations with one or more tainted operands.

#### Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

#### Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of `0` and `-1`. Check both operands for negative values.

#### Example — Modulo with User Input

```
#include <stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem =  128%userden;  //Noncompliant
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using a user input. The input is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

**Correction — Check Operand Values**

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
#include<stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem = 0;
    if (userden > 0 ) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

## Check Information

**Group:** Rec. 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

INT10-C

# CERT C: Rec. INT12-C

Do not make assumptions about the type of a plain int bit-field when used in an expression

## Description

### Rule Definition

*Do not make assumptions about the type of a plain int bit-field when used in an expression.*

### Polyspace Implementation

The rule checker checks for **Bit-field declared without appropriate type**.

## Examples

### Bit-field declared without appropriate type

#### Issue

The issue occurs when you declare a bit-filed without an appropriate type.

#### Risk

Using `int` is implementation-defined because bit-fields of type `int` can be either `signed` or `unsigned`.

The use of `enum`, `short char`, or any other type of bit-field is not permitted in C90 because the behavior is undefined.

In C99, the implementation can potentially define other integer types that are permitted in bit-field declarations.

## Check Information

**Group:** Rec. 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

INT12-C

# CERT C: Rec. INT13-C

Use bitwise operators only on unsigned operands

## Description

### Rule Definition

*Use bitwise operators only on unsigned operands.*

### Polyspace Implementation

The rule checker checks for **Bitwise operation on negative value**.

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Bitwise operation on negative value

**Issue**

**Bitwise operation on negative value** detects bitwise operators (>>, ^, `|`, ~, `but`, `not`, &) used on signed integer variables with negative values.

**Risk**

If the value of the signed integer is negative, bitwise operation results can be unexpected because:

- Bitwise operations on negative values are compiler-specific.
- Unexpected calculations can lead to additional vulnerabilities, such as buffer overflow.

**Fix**

When performing bitwise operations, use `unsigned` integers to avoid unexpected results.

**Example - Right-Shift of Negative Integer**

```
#include <stdio.h>
#include <stdarg.h>

static void demo_sprintf(const char *format, ...)
{
    int rc;
    va_list ap;
    char buf[sizeof("256")];

    va_start(ap, format);
    rc = vsprintf(buf, format, ap);
    if (rc == -1 || rc >= sizeof(buf)) {
        /* Handle error */
    }
    va_end(ap);
}

void bug_bitwiseneg()
{
    int stringify = 0x80000000;
    demo_sprintf("%u", stringify >> 24); //Noncompliant
}
```

In this example, the statement `demo_sprintf("%u", stringify >> 24)` stops the program unexpectedly. You expect the result of `stringify >> 24` to be `0x80`. However, the actual result is `0xffffff80` because `stringify` is signed and negative. The sign bit is also shifted.

#### Correction — Add `unsigned` Keyword

By adding the `unsigned` keyword, `stringify` is not negative and the right-shift operation gives the expected result of `0x80`.

```
#include <stdio.h>
#include <stdarg.h>

static void demo_sprintf(const char *format, ...)
{
    int rc;
    va_list ap;
    char buf[sizeof("256")];

    va_start(ap, format);
    rc = vsprintf(buf, format, ap);
    if (rc == -1 || rc >= sizeof(buf)) {
        /* Handle error */
    }
    va_end(ap);
}

void corrected_bitwiseneg()
{
    unsigned int stringify = 0x80000000;
    demo_sprintf("%u", stringify >> 24);
}
```

## Check Information

**Group:** Rec. 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

INT13-C

# CERT C: Rec. INT14-C

Avoid performing bitwise and arithmetic operations on the same data

## Description

### Rule Definition

*Avoid performing bitwise and arithmetic operations on the same data.*

### Polyspace Implementation

The rule checker checks for **Bitwise and arithmetic operation on the same data**.

## Examples

### Bitwise and arithmetic operation on the same data

#### Issue

**Bitwise and arithmetic operation on a same data** detects statements with bitwise and arithmetic operations on the same variable or expression.

#### Risk

Mixed bitwise and arithmetic operations *do* compile. However, the size of integer types affects the result of these mixed operations. Mixed operations also reduce readability and maintainability.

#### Fix

Separate bitwise and arithmetic operations, or use only one type of operation per statement.

#### Example - Shift and Addition

```
unsigned int bitwisearithmix()
{
    unsigned int var = 50;
    var += (var << 2) + 1; //Noncompliant
    return var;
}
```

This example shows bitwise and arithmetic operations on the variable `var`. `var` is shifted by two (bitwise), then increased by 1 and added to itself (arithmetic).

#### Correction — Arithmetic Operations Only

You can reduce this expression to arithmetic-only operations: `var + (var << 2)` is equivalent to `var * 5`.

```
unsigned int bitwisearithmix()
{
    unsigned int var = 50;
    var = var * 5 +1;
    return var;
}
```

## Check Information

**Group:** Rec. 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

INT14-C

# CERT C: Rec. INT18-C

Evaluate integer expressions in a larger size before comparing or assigning to that size

## Description

### Rule Definition

*Evaluate integer expressions in a larger size before comparing or assigning to that size.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Integer overflow**.
- **Unsigned integer overflow**.

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Integer overflow

#### Issue

**Integer overflow** occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer overflows on signed integers result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.

- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;              //Noncompliant
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

#### Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

### Unsigned integer overflow

#### Issue

**Unsigned integer overflow** occurs when an operation on unsigned integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

**Risk**

The C11 standard states that unsigned integer overflows result in wrap-around behavior. However, a wrap around behavior might not always be desirable. For instance, if the result of a computation is used as an array size and the computation overflows, the array size is much smaller than expected.

**Fix**

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling. In the error handling code, you can override the default wrap-around behavior for overflows and implement saturation behavior, for instance.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Add One to Maximum Unsigned Integer**

```
#include <limits.h>

unsigned int plusplus(void) {

    unsigned uvar = UINT_MAX;
    uvar++; //Noncompliant
    return uvar;
}
```

In the third statement of this function, the variable `uvar` is increased by 1. However, the value of `uvar` is the maximum unsigned integer value, so 1 plus the maximum integer value cannot be represented by an `unsigned int`. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `uvar` is reduced by modulo `UINT_MAX`. The result is `uvar = 1`.

**Correction — Different Storage Type**

One possible correction is to store the operation result in a larger data type. In this example, by returning an `unsigned long long` instead of an `unsigned int`, the overflow error is fixed.

```
#include <limits.h>
```

```
unsigned long long plusplus(void) {

    unsigned long long ullvar = UINT_MAX;
    ullvar++;
    return ullvar;
}
```

## Check Information

**Group:** Rec. 04. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

INT18-C

# CERT C: Rec. FLP00-C

Understand the limitations of floating-point numbers

## Description

### Rule Definition

*Understand the limitations of floating-point numbers.*

### Polyspace Implementation

The rule checker checks for **Absorption of float operand**.

## Examples

### Absorption of float operand

#### Issue

**Absorption of float operand** occurs when one operand of an addition or subtraction operation is *always* negligibly small compared to the other operand. Therefore, the result of the operation is always equal to the value of the larger operand, making the operation redundant.

#### Risk

Redundant operations waste execution cycles of your processor.

The absorption of a float operand can indicate design issues elsewhere in the code. It is possible that the developer expected a different range for one of the operands and did not expect the redundancy of the operation. However, the operand range is different from what the developer expects because of issues elsewhere in the code.

#### Fix

See if the operand ranges are what you expect. To see the ranges, place your cursor on the operation.

- If the ranges are what you expect, justify why you have the redundant operation in place. For instance, the code is only partially written and you anticipate other values for one or both of the operands from future unwritten code.

  If you cannot justify the redundant operation, remove it.
- If the ranges are not what you expect, in your code, trace back to see where the ranges come from. To begin your traceback, search for instances of the operand in your code. Browse through previous instances of the operand and determine where the unexpected range originates.

To determine when one operand is negligible compared to the other operand, the defect uses rules based on IEEE 754 standards. To fix the defect, instead of using the actual rules, you can use this heuristic: the ratio of the larger to the smaller operand must be less than $2^{p-1}$ at least for some values. Here, `p` is equal to 24 for 32-bit precision and 53 for 64-bit precision. To determine the precision, the defect uses your specification for `Target processor type (-target)`.

### Example - One Addition Operand Negligibly Smaller Than The Other Operand

```
#include <stdlib.h>

float get_signal(void);
void do_operation(float);

float input_signal1(void) {
    float temp = get_signal();
    if(temp > 0. && temp < 1e-30)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

float input_signal2(void) {
    float temp = get_signal();
    if(temp > 1.)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

void main() {
    float signal1 = input_signal1();
    float signal2 = input_signal2();
    float super_signal = signal1 + signal2; //Noncompliant
    do_operation(super_signal);
}
```

In this example, the defect appears on the addition because the operand `signal1` is in the range `(0,1e-30)` but `signal2` is greater than `1`.

### Correction — Remove Redundant Operation

One possible correction is to remove the redundant addition operation. In the following corrected code, the operand `signal2` and its associated code is also removed from consideration.

```
#include <stdlib.h>

float get_signal(void);
void do_operation(float);

float input_signal1(void) {
    float temp = get_signal();
    if(temp > 0. && temp < 1e-30)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

void main() {
    float signal1 = input_signal1();
    do_operation(signal1);
}
```

#### Correction — Verify Operand Range

Another possible correction is to see if the operand ranges are what you expect. For instance, if one of the operand range is not supposed to be negligibly small, fix the issue causing the small range. In the following corrected code, the range `(0,1e-2)` is imposed on `signal2` so that it is not *always* negligibly small as compared to `signal1`.

```
#include <stdlib.h>

float get_signal(void);
void do_operation(float);

float input_signal1(void) {
    float temp = get_signal();
    if(temp > 0. && temp < 1e-2)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

float input_signal2(void) {
    float temp = get_signal();
    if(temp > 1.)
        return temp;
    else {
       /* Reject value */
       exit(EXIT_FAILURE);
    }
}

void main() {
    float signal1 = input_signal1();
    float signal2 = input_signal2();
    float super_signal = signal1 + signal2;
    do_operation(super_signal);
}
```

## Check Information

**Group:** Rec. 05. Floating Point (FLP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

“Check for and Review Coding Standard Violations”

**External Websites**
FLP00-C

# CERT C: Rec. FLP02-C

Avoid using floating-point numbers when precise computation is needed

## Description

### Rule Definition

*Avoid using floating-point numbers when precise computation is needed.*

### Polyspace Implementation

The rule checker checks for **Floating point comparison with equality operators**.

## Examples

### Floating point comparison with equality operators

#### Issue

**Floating point comparison with equality operators** occurs when you use an equality (==) or inequality (!=) operation with floating-point numbers.

Polyspace does not raise a defect for an equality or inequality operation with floating-point numbers when:

- The comparison is between two float constants.

  ```
      float flt = 1.0;
      if (flt == 1.1)
  ```
- The comparison is between a constant and a variable that can take a finite, reasonably small number of values.

  ```
  float x;

  int rand = random();
  switch(rand) {
  case 1: x = 0.0; break;
  case 2: x = 1.3; break;
  case 3: x = 1.7; break;
  case 4: x = 2.0; break;
  default: x = 3.5; break; }
  //…
  if (x==1.3)
  ```
- The comparison is between floating-point expressions that contain only integer values.

  ```
  float x = 0.0;
  for (x=0.0;x!=100.0;x+=1.0) {
  //…
  if (random) break;
  }

  if (3*x+4==2*x-1)
  ```

```
//…
if (3*x+4 == 1.3)
```

- One of the operands is `0.0`, unless you use the option flag `-detect-bad-float-op-on-zero`.

  ```
  /* Defect detected when
  you use the option flag */

  if (x==0.0f)
  ```

  If you are running an analysis through the user interface, you can enter this option in the **Other** field, under the **Advanced Settings** node on the **Configuration** pane. See `Other`.

  At the command line, add the flag to your analysis command.

  ```
  polyspace-bug-finder -sources filename ^
  -checkers BAD_FLOAT_OP -detect-bad-float-op-on-zero
  ```

#### Risk

Checking for equality or inequality of two floating-point values might return unexpected results because floating-point representations are inexact and involve rounding errors.

#### Fix

Instead of checking for equality of floating-point values:

```
if (val1 == val2)
```

check if their difference is less than a predefined tolerance value (for instance, the value `FLT_EPSILON` defined in `float.h`):

```
#include <float.h>
if(fabs(val1-val2) < FLT_EPSILON)
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Floats Inequality in `for`-loop

```
#include <stdio.h>
#include <math.h>
#include <float.h>

void func(void)
{
    float f;
    for (f = 1.0; f != 2.0; f = f + 0.1)     //Noncompliant
        (void)printf("Value: %f\n", f);
}
```

In this function, the `for`-loop tests the inequality of `f` and the number 2.0 as a stopping mechanism. The number of iterations is difficult to determine, or might be infinite, because of the imprecision in floating-point representation.

#### Correction — Change the Operator

One possible correction is to use a different operator that is not as strict. For example, an inequality like >= or <=.

```
#include <stdio.h>
#include <math.h>
#include <float.h>

void func(void)
{
    float f;
    for (f = 1.0; f <= 2.0; f = f + 0.1)
        (void)printf("Value: %f\n", f);
}
```

## Check Information

**Group:** Rec. 05. Floating Point (FLP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

FLP02-C

# CERT C: Rec. FLP03-C

Detect and handle floating-point errors

## Description

### Rule Definition

*Detect and handle floating-point errors.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Float conversion overflow**.
- **Float overflow**.
- **Float division by zero**.

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Float conversion overflow

#### Issue

**Float conversion overflow** occurs when converting a floating point number to a smaller floating point data type. If the variable does not have enough memory to represent the original number, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing conversion in subsequent computations and do not account for the overflow, you can see unexpected results.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being converted acquires its current value You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller floating point types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and `NaNs`. Operations that results in infinities and `NaNs` might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats)`.

#### Example - Converting from double to float

```
float convert(void) {

    double diam = 1e100;
    return (float)diam; //Noncompliant
}
```

In the return statement, the variable `diam` of type double (64 bits) is converted to a variable of type float (32 bits). However, the value 1^100 requires more than 32 bits to be precisely represented.

### Float overflow

#### Issue

**Float overflow** occurs when an operation on floating point variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing computation in subsequent computations and do not account for the overflow, you can see unexpected results.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and `NaNs`. Operations that results in infinities and `NaNs` might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats)`.

#### Example - Multiplication of Floats

```
#include <float.h>

float square(void) {

   float val = FLT_MAX;
   return val * val;   //Noncompliant
}
```

In the return statement, the variable `val` is multiplied by itself. The square of the maximum float value cannot be represented by a float (the return type for this function) because the value of `val` is the maximum float value.

#### Correction — Different Storage Type

One possible correction is to store the result of the operation in a larger data type. In this example, by returning a `double` instead of a `float`, the overflow defect is fixed.

```
#include <float.h>

double square(void) {
    float val = FLT_MAX;

    return (double)val * (double)val;
}
```

### Float division by zero

#### Issue

**Float division by zero** occurs when the denominator of a division operation can be a zero-valued floating point number.

#### Risk

A division by zero can result in a program crash.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the denominator variable acquires a zero value. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and `NaN`s. Operations that results in infinities and `NaN`s might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats)`.

#### Example - Dividing a Floating Point Number by Zero

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    result = num/denom; //Noncompliant

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

#### Correction — Check Before Division

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    if( ((int)denom) != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If denom is always zero, this correction can produce a dead code defect in your Polyspace results.

#### Correction — Change Denominator

One possible correction is to change the denominator value so that denom is not zero.

```
float fraction(float num)
{
    float denom = 2.0;
    float result = 0.0;

    result = num/denom;

    return result;
}
```

## Check Information

**Group:** Rec. 05. Floating Point (FLP)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C (-cert-c)

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FLP03-C

# CERT C: Rec. FLP06-C

Convert integers to floating point for floating-point operations

## Description

### Rule Definition

*Convert integers to floating point for floating-point operations.*

### Polyspace Implementation

The rule checker checks for **Float overflow**.

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Float overflow

#### Issue

**Float overflow** occurs when an operation on floating point variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing computation in subsequent computations and do not account for the overflow, you can see unexpected results.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and NaNs. Operations that results in infinities and NaNs might be flagged as defects. To handle infinities and NaN values in your code, use the option Consider non finite floats (-allow-non-finite-floats).

#### Example - Multiplication of Floats

```
#include <float.h>

float square(void) {

   float val = FLT_MAX;
   return val * val;   //Noncompliant
}
```

In the return statement, the variable val is multiplied by itself. The square of the maximum float value cannot be represented by a float (the return type for this function) because the value of val is the maximum float value.

#### Correction — Different Storage Type

One possible correction is to store the result of the operation in a larger data type. In this example, by returning a double instead of a float, the overflow defect is fixed.

```
#include <float.h>

double square(void) {
    float val = FLT_MAX;

    return (double)val * (double)val;
}
```

## Check Information

**Group:** Rec. 05. Floating Point (FLP)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C (-cert-c)

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FLP06-C

# CERT C: Rec. ARR01-C

Do not apply the sizeof operator to a pointer when taking the size of an array

## Description

### Rule Definition

*Do not apply the sizeof operator to a pointer when taking the size of an array.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Wrong type used in sizeof**.
- **Possible misuse of sizeof**.

## Examples

### Wrong type used in sizeof

#### Issue

**Wrong type used in sizeof** occurs when both of the following conditions hold:

- You assign the address of a block of memory to a pointer, or transfer data between two blocks of memory. The assignment or copy uses the `sizeof` operator.

  For instance, you initialize a pointer using `malloc(sizeof(`*`type`*`))` or copy data between two addresses using `memcpy(`*`destination_ptr`*`, `*`source_ptr`*`, sizeof(`*`type`*`))`.
- You use an incorrect type as argument of the `sizeof` operator. You use the pointer type instead of the type that the pointer points to.

  For instance, to initialize a *`type`*`*` pointer, you use `malloc(sizeof(`*`type`*`*))` instead of `malloc(sizeof(`*`type`*`))`.

#### Risk

Irrespective of what *`type`* stands for, the expression `sizeof(`*`type`*`*)` always returns a fixed size. The size returned is the pointer size on your platform in bytes. The appearance of `sizeof(`*`type`*`*)` often indicates an unintended usage. The error can cause allocation of a memory block that is much smaller than what you need and lead to weaknesses such as buffer overflows.

For instance, assume that `structType` is a structure with ten `int` variables. If you initialize a `structType*` pointer using `malloc(sizeof(structType*))` on a 32-bit platform, the pointer is assigned a memory block of four bytes. However, to be allocated completely for one `structType` variable, the `structType*` pointer must point to a memory block of `sizeof(structType) = 10 * sizeof(int)` bytes. The required size is much greater than the actual allocated size of four bytes.

#### Fix

To initialize a *`type`*`*` pointer, replace `sizeof(`*`type`*`*)` in your pointer initialization expression with `sizeof(`*`type`*`)`.

#### Example - Allocate a Char Array With sizeof

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char*) * 5); //Noncompliant
    free(str);

}
```

In this example, memory is allocated for the character pointer `str` using a `malloc` of five char pointers. However, `str` is a pointer to a character, not a pointer to a character pointer. Therefore the `sizeof` argument, `char*`, is incorrect.

#### Correction — Match Pointer Type to sizeof Argument

One possible correction is to match the argument to the pointer type. In this example, `str` is a character pointer, therefore the argument must also be a character.

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char) * 5);
    free(str);

}
```

### Possible misuse of sizeof

#### Issue

**Possible misuse of sizeof** occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

#### Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

**Fix**

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

**Example - `sizeof` Used Incorrectly to Determine Array Size**

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    { //Noncompliant
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

**Correction — Determine Array Size in Another Way**

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

## Check Information

**Group:** Rec. 06. Arrays (ARR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

ARR01-C

# CERT C: Rec. ARR02-C

Explicitly specify array bounds, even if implicitly defined by an initializer

## Description

### Rule Definition

*Explicitly specify array bounds, even if implicitly defined by an initializer.*

### Polyspace Implementation

The rule checker checks for the issue **Improper array initialization**.

## Examples

### Improper array initialization

#### Issue

**Improper array initialization** occurs when Polyspace Bug Finder considers that an array initialization using initializers is incorrect.

This defect applies to normal and designated initializers. In C99, with designated initializers, you can place the elements of an array initializer in any order and implicitly initialize some array elements. The designated initializers use the array index to establish correspondence between an array element and an array initializer element. For instance, the statement `int arr[6] = { [4] = 29, [2] = 15 }` is equivalent to `int arr[6] = { 0, 0, 15, 0, 29, 0 }`.

You can use initializers incorrectly in one of the following ways.

| Issue | Risk | Possible Fix |
|---|---|---|
| In your initializer for a one-dimensional array, you have more elements than the array size. | Unused array initializer elements indicate a possible coding error. | Increase the array size or remove excess elements. |
| You place the braces enclosing initializer values incorrectly. | Because of the incorrect placement of braces, some array initializer elements are not used.<br>Unused array initializer elements indicate a possible coding error. | Place braces correctly. |
| In your designated initializer, you do not initialize the first element of the array explicitly. | The implicit initialization of the first array element indicates a possible coding error. You possibly overlooked the fact that array indexing starts from 0. | Initialize all elements explicitly. |

| Issue | Risk | Possible Fix |
|---|---|---|
| In your designated initializer, you initialize an element twice. | The first initialization is overridden.<br>The redundant first initialization indicates a possible coding error. | Remove the redundant initialization. |
| You use designated and nondesignated initializers in the same initialization. | You or another reviewer of your code cannot determine the size of the array by inspection. | Use either designated or nondesignated initializers. |

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Incorrectly Placed Braces (C Only)

```
int arr[2][3]
= {{1, 2},
    {3, 4},
    {5, 6} //Noncompliant
};
```

In this example, the array `arr` is initialized as `{1,2,0,3,4,0}`. Because the initializer contains `{5,6}`, you might expect the array to be initialized `{1,2,3,4,5,6}`.

#### Correction — Place Braces Correctly

One possible correction is to place the braces correctly so that all elements are explicitly initialized.

```
int a1[2][3]
= {{1, 2, 3},
    {4, 5, 6}
};
```

#### Example - First Element Not Explicitly Initialized

```
int arr[5]
= {
    [1] = 2,
    [2] = 3,
    [3] = 4,
    [4] = 5
};               //Noncompliant
```

In this example, `arr[0]` is not explicitly initialized. It is possible that the programmer did not consider that the array indexing starts from 0.

#### Correction — Explicitly Initialize All Elements

One possible correction is to initialize all elements explicitly.

```
int arr[5]
= {
    [0] = 1,
    [1] = 2,
    [2] = 3,
    [3] = 4,
    [4] = 5
};
```

#### Example - Element Initialized Twice

```
int arr[5]
= {
    [0] = 1,
    [1] = 2,
    [2] = 3,
    [2] = 4, //Noncompliant
    [4] = 5
};
```

In this example, `arr[2]` is initialized twice. The first initialization is overridden. In this case, because `arr[3]` was not explicitly initialized, it is possible that the programmer intended to initialize `arr[3]` when `arr[2]` was initialized a second time.

#### Correction — Fix Redundant Initialization

One possible correction is to eliminate the redundant initialization.

```
int arr[5]
= {
    [0] = 1,
    [1] = 2,
    [2] = 3,
    [3] = 4,
    [4] = 5
};
```

#### Example - Mix of Designated and Nondesignated Initializers

```
int arr[]
= {
    [0] = 1,
    [3] = 3,
    4,
    [5] = 5,
    6
    };               //Noncompliant
```

In this example, because a mix of designated and nondesignated initializers are used, it is difficult to determine the size of `arr` by inspection.

#### Correction — Use Only Designated Initializers

One possible correction is to use only designated initializers for array initialization and to specify the size of the array explicitly.

```
int arr[7]
= {
    [0] = 1,
    [3] = 3,
    [4] = 4,
    [5] = 5,
    [6] = 6
};
```

## Check Information

**Group:** Rec. 06. Arrays (ARR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ARR02-C

# CERT C: Rec. STR02-C

Sanitize data passed to complex subsystems

## Description

### Rule Definition

*Sanitize data passed to complex subsystems.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Execution of externally controlled command**.
- **Command executed from externally controlled path**.
- **Library loaded from externally controlled path**.

### Extend Checker

A default Bug Finder analysis might not flag this checker for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Execution of externally controlled command

#### Issue

**Execution of externally controlled command** checks for commands that are fully or partially constructed from externally controlled input.

#### Risk

Attackers can use the externally controlled input as operating system commands, or arguments to the application. An attacker could read or modify sensitive data can be read or modified, execute unintended code, or gain access to other aspects of the program.

#### Fix

Validate the inputs to allow only intended input values. For example, create a list of acceptable inputs and compare the input against this list.

#### Example - Call External Command

```
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "unistd.h"
#include "dlfcn.h"
#include "limits.h"
#define MAX 128
```

```
void taintedexternalcmd(void)
{
    char* usercmd;
    fgets(usercmd,MAX,stdin);
    char cmd[MAX] = "/usr/bin/cat ";
    strcat(cmd, usercmd);
    system(cmd);  //Noncompliant
}
```

This example function calls a command from a user input without checking the command variable.

**Correction — Use a Predefined Command**

One possible correction is to use a `switch` statement to run a predefined command, using the user input as the switch variable.

```
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "unistd.h"
#include "dlfcn.h"
#include "limits.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
enum { CMD0 = 1, CMD1, CMD2 };

void taintedexternalcmd(void)
{
    int usercmd = strtol(getenv("cmd"),NULL,10);
    char cmd[SIZE128] = "/usr/bin/cat ";

    switch(usercmd) {
        case CMD0:
            strcat(cmd, "*.c");
            break;
        case CMD1:
            strcat(cmd, "*.h");
            break;
        case CMD2:
            strcat(cmd, "*.cpp");
            break;
        default:
            strcat(cmd, "*.c");
    }
    system(cmd);
}
```

### Command executed from externally controlled path

#### Issue

**Command executed from externally controlled path** checks the path of commands that the application controls. If the path of a command is from or constructed from external sources, Bug Finder flags the command function.

#### Risk

An attacker can:

- Change the command that the program executes, possibly to a command that only the attack can control.
- Change the environment in which the command executes, by which the attacker controls what the command means and does.

#### Fix

Before calling the command, validate the path to make sure that it is the intended location.

#### Example - Executing Path from Environment Variable

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedpathcmd() {
    char cmd[SIZE128] = "";
    char* userpath = getenv("MYAPP_PATH");

    strncpy(cmd, userpath, SIZE100);
    strcat(cmd, "/ls *");
    /* Launching command */
    system(cmd);  //Noncompliant
}
```

This example obtains a path from an environment variable `MYAPP_PATH`. `system` runs a command from that path without checking the value of the path. If the path is not the intended path, your program executes in the wrong location.

#### Correction — Use Trusted Path

One possible correction is to use a list of allowed paths to match against the environment variable path.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

/* Function to sanitize a string */
int sanitize_str(char* s, size_t n) {
    int res = 0;
    /* String is ok if */
    if (s && n>0 && n<SIZE128) {
        /* - string is not null                     */
        /* - string has a positive and limited size */
        s[n-1] = '\0';  /* Add a security \0 char at end of string */
        /* Tainted pointer detected above, used as "firewall" */
        res = 1;
    }
    return res;
}

/* Authorized path ids */
enum { PATH0=1, PATH1, PATH2 };

void taintedpathcmd() {
    char cmd[SIZE128] = "";

    char* userpathid = getenv("MYAPP_PATH_ID");
    if (sanitize_str(userpathid, SIZE100)) {
        int pathid = atoi(userpathid);


        char path[SIZE128] = "";
        switch(pathid) {
            case PATH0:
                strcpy(path, "/usr/local/my_app0");
                break;
            case PATH1:
                strcpy(path, "/usr/local/my_app1");
                break;
            case PATH2:
                strcpy(path, "/usr/local/my_app2");
                break;
            default:
                /* do nothing */
        break;
        }
        if (strlen(path)>0) {
            strncpy(cmd, path, SIZE100);
            strcat(cmd, "/ls *");
            system(cmd);
        }
    }
}
```

### Library loaded from externally controlled path

#### Issue

**Library loaded from externally controlled path** looks for libraries loaded from fixed or controlled paths. If unintended actors can control one or more locations on this fixed path, Bug Finder raises a defect.

#### Risk

If an attacker knows or controls the path that you use to load a library, the attacker can change:

- The library that the program loads, replacing the intended library and commands.
- The environment in which the library executes, giving unintended permissions and capabilities to the attacker.

#### Fix

When possible, use hard-coded or fully qualified path names to load libraries. It is possible the hard-coded paths do not work on other systems. Use a centralized location for hard-coded paths, so that you can easily modify the path within the source code.

Another solution is to use functions that require explicit paths. For example, `system()` does not require a full path because it can use the `PATH` environment variable. However, `execl()` and `execv()` do require the full path.

#### Example - Call Custom Library

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void* taintedpathlib() {
    void* libhandle = NULL;
    char lib[SIZE128] = "";
    char* userpath = getenv("LD_LIBRARY_PATH");
    strncpy(lib, userpath, SIZE128);
    strcat(lib, "/libX.so");
    libhandle = dlopen(lib, 0x00001); //Noncompliant
    return libhandle;
}
```

This example loads the library `libX.so` from an environment variable `LD_LIBRARY_PATH`. An attacker can change the library path in this environment variable. The actual library you load could be a different library from the one that you intend.

#### Correction — Change and Check Path

One possible correction is to change how you get the library path and check the path of the library before opening the library. This example receives the path as an input argument. Then the path is checked to make sure the library is not under `/usr/`.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

/* Function to sanitize a string */
int sanitize_str(char* s, size_t n) {
    /* strlen is used here as a kind of firewall for tainted string errors */
    int res = (strlen(s) > 0 && strlen(s) < n);
    return res;
}
void* taintedpathlib(char* userpath) {
    void* libhandle = NULL;
    if (sanitize_str(userpath, SIZE128)) {
        char lib[SIZE128] = "";

        if (strncmp(userpath, "/usr", 4)!=0) {
            strncpy(lib, userpath, SIZE128);
            strcat(lib, "/libX.so");
            libhandle = dlopen(lib, RTLD_LAZY);
        }
    }
    return libhandle;
}
```

## Check Information

**Group:** Rec. 07. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

STR02-C

# CERT C: Rec. STR03-C

Do not inadvertently truncate a string

## Description

### Rule Definition

*Do not inadvertently truncate a string.*

### Polyspace Implementation

The rule checker checks for **Invalid use of standard library string routine**.

## Examples

### Invalid use of standard library string routine

#### Issue

**Invalid use of standard library string routine** occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source);
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Use of Standard Library String Routine Error

```
#include <string.h>
#include <stdio.h>
```

```
 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);  //Noncompliant
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

**Correction — Use Valid Arguments**

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

## Check Information

**Group:** Rec. 07. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

**Topics**

“Check for and Review Coding Standard Violations”

**External Websites**

STR03-C

# CERT C: Rec. STR07-C

Use the bounds-checking interfaces for string manipulation

## Description

### Rule Definition

*Use the bounds-checking interfaces for string manipulation.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Use of dangerous standard function**.
- **Destination buffer overflow in string manipulation**.
- **Insufficient destination buffer size**.

## Examples

### Use of dangerous standard function

#### Issue

This issue occurs when your code uses standard functions that write data to a buffer in a way that can result in buffer overflows.

The following table lists dangerous standard functions, the risks of using each function, and what function to use instead. The checker flags:

- Any use of an inherently dangerous function.
- An use of a possibly dangerous function only if the size of the buffer to which data is written can be determined at compile time. The checker does not flag an use of such a function with a dynamically allocated buffer.

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `gets` | Inherently dangerous — You cannot control the length of input from the console. | `fgets` |
| `cin` | Inherently dangerous — You cannot control the length of input from the console. | Avoid or prefaces calls to `cin` with `cin.width`. |
| `strcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `strncpy` |

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `stpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `stpncpy` |
| `lstrcpy` or `StrCpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy` |
| `strcat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `strncat`, `strlcat`, or `strcat_s` |
| `lstrcat` or `StrCat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat` |
| `wcpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcpncpy` |
| `wcscat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `wcsncat`, `wcslcat`, or `wcncat_s` |
| `wcscpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcsncpy` |
| `sprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `snprintf` |
| `vsprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `vsnprintf` |

#### Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.

- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Using `sprintf`

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1) //Noncompliant
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

#### Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

### Destination buffer overflow in string manipulation

#### Issue

**Destination buffer overflow in string manipulation** occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string); //Noncompliant
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

#### Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Insufficient destination buffer size

#### Issue

**Insufficient destination buffer size** occurs when the destination buffer in a `strcpy` operation cannot accommodate the source buffer and a null terminator. This issue is reported if the size of the source buffer is unknown. Consider this code:

```
int main (int argc, char *argv[])
{
  const char *const input = ((argc && argv[0]) ? argv[0] : "");
  char str[100];
  strcpy(input, str); // Noncompliant
}
```

In this case, the size of the source buffer `input` is unknown. The size of the destination buffer `str` might be smaller than the value `(strlen(input)+1)`. Polyspace reports a violation on the `strcpy` operation.

**Risk**

Using a destination buffer of insufficient size might allow an attacker to cause a buffer overflow. In the preceding code example, if `argv[0]` contains 100 or more characters, the `strcpy` operation results in a buffer overflow.

**Fix**

Before calling the function `strcpy()`, allocate sufficient memory dynamically. For instance, use the function `strlen()` to determine the size of the source buffer and then allocate the destination buffer so that its size is greater than the value `strlen(source) + 1`.

**Example — Destination Buffer Too Small**

In this example, the size of the `source` buffer is unknown, while the size of the `destination` buffer is fixed at `128`. The size of the `destination` buffer might not be sufficient to accommodate the characters from the `source` buffer and terminate the buffer with a null. Polyspace reports a violation of the rule.

```
#include <string.h>

int main(int argc, char *argv[]) {
  const char *const source = (argc && argv[0]) ? argv[0] : "";
  char destination[128];
  strcpy(source, destination);//Noncompliant

  return 0;
}
```

**Correction — Allocate Sufficient Memory for Destination Buffer**

This violation is resolved by allocating sufficient memory for the `destination` buffer. For instance, use the function `strlen()` to calculate the size of the `source` buffer and allocate sufficient memory for the `destination` buffer so that it can accommodate all characters from the `source` buffer and the null terminator (`'\0'` ).

```
#include <string.h>

int main(int argc, char *argv[]) {
  const char *const source = (argc && argv[0]) ? argv[0] : "";
  char* destination = (char *)malloc(strlen(source)+ 1);
  if(destination!=NULL){
      strcpy(source, destination);//Compliant
  }else{
      /*Handle Error*/
  }
  //...
  free(destination);
  return 0;
}
```

## Check Information

**Group:** Rec. 07. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

STR07-C

# CERT C: Rec. STR11-C

Do not specify the bound of a character array initialized with a string literal

## Description

### Rule Definition

*Do not specify the bound of a character array initialized with a string literal.*

### Polyspace Implementation

The rule checker checks for **Missing null in string array**.

## Examples

### Missing null in string array

#### Issue

**Missing null in string array** occurs when a string does not have enough space to terminate with a null character `'\0'`.

This defect applies only for projects in C.

#### Risk

A buffer overflow can occur if you copy a string to an array without assuming the implicit null terminator.

#### Fix

If you initialize a character array with a literal, avoid specifying the array bounds.

```
char three[]  = "THREE";
```

The compiler automatically allocates space for a null terminator. In the preceding example, the compiler allocates sufficient space for five characters and a null terminator.

If the issue occurs after initialization, you might have to increase the size of the array by one to account for the null terminator.

In certain circumstances, you might want to initialize the character array with a sequence of characters instead of a string. In this situation, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Array size is too small**

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE"; //Noncompliant
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

**Correction — Increase Array Size**

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

**Correction — Change Initialization Method**

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

## Check Information

**Group:** Rec. 07. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

STR11-C

# CERT C: Rec. MEM00-C

Allocate and free memory in the same module, at the same level of abstraction

## Description

### Rule Definition

*Allocate and free memory in the same module, at the same level of abstraction.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Invalid free of pointer**.
- **Deallocation of previously deallocated pointer**.
- **Use of previously freed pointer**.

## Examples

### Invalid free of pointer

#### Issue

**Invalid free of pointer** occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

#### Risk

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function or a previous `malloc` function on a different pointer.

#### Fix

In most cases, you can fix the issue by removing the `free` statement. If the pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

#### Example - Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);    //Noncompliant
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer p is deallocated using the free function. However, p points to a memory location that was not dynamically allocated.

#### Correction — Remove Pointer Deallocation

If the number of elements of the array p is known at compile time, one possible correction is to remove the deallocation of the pointer p.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

#### Correction — Introduce Pointer Allocation

If the number of elements of the array p is not known at compile time, one possible correction is to dynamically allocate memory to the array p.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

### Deallocation of previously deallocated pointer

#### Issue

**Deallocation of previously deallocated pointer** occurs when a block of memory is freed more than once using the free function without an intermediate allocation.

#### Risk

When a pointer is allocated dynamic memory with malloc, calloc or realloc, it points to a memory location on the heap. When you use the free function on this pointer, the associated block of memory is freed for reallocation. Trying to free this block of memory can result in a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. See if you intended to allocate a memory block to the pointer between the first deallocation and the second. Otherwise, remove the second `free` statement.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before freeing pointers, check them for NULL values and handle the error. In this way, you are protected against freeing an already freed block.

#### Example - Deallocation of Previously Deallocated Pointer Error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);        //Noncompliant
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

#### Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
 }
```

### Use of previously freed pointer

#### Issue

**Use of previously freed pointer** occurs when you access a block of memory after freeing the block using the `free` function.

#### Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

**Fix**

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

**Example - Use of Previously Freed Pointer Error**

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift; //Noncompliant
    /* Defect: Reading a freed pointer */

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

**Correction — Free Pointer After Use**

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

## Check Information

**Group:** Rec. 08. Memory Management (MEM)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MEM00-C

# CERT C: Rec. MEM01-C

Store a new value in pointers immediately after free()

## Description

### Rule Definition

*Store a new value in pointers immediately after free().*

### Polyspace Implementation

The rule checker checks for **Missing reset of a freed pointer**.

## Examples

### Missing reset of a freed pointer

#### Issue

**Missing reset of a freed pointer** detects pointers that have been freed and not reassigned another value. After freeing a pointer, the memory data is still accessible. To clear this data, the pointer must also be set to NULL or another value.

#### Risk

Not resetting pointers can cause dangling pointers. Dangling pointers can cause:

- Freeing already freed memory.
- Reading from or writing to already freed memory.
- Hackers executing code stored in freed pointers or with vulnerable permissions.

#### Fix

After freeing a pointer, if it is not immediately assigned to another valid address, set the pointer to NULL.

#### Example - Free Without Reset

```
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};

void missingfreedptrreset()
{
    static char *str = NULL;

    if (str == NULL)
        str = (char *)malloc(SIZE20);

    if (str != NULL)
```

```
        free(str); //Noncompliant
}
```

In this example, the pointer `str` is freed at the end of the program. The next call to `bug_missingfreedptrrese` can fail because `str` is not NULL and the initialization to NULL can be invalid.

#### Correction — Redefine `free` to Free and Reset

One possible correction is to customize `free` so that when you free a pointer, it is automatically reset.

```
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};

static void sanitize_free(void **p)
{
    if ((p != NULL) && (*p != NULL))
    {
        free(*p);
        *p = NULL;
    }
}

#define free(X) sanitize_free((void **)&X)

void missingfreedptrreset()
{
    static char *str = NULL;

    if (str == NULL)
        str = (char *)malloc(SIZE20);

    if (str != ((void *)0))
    {
        free(str);
    }
}
```

## Check Information

**Group:** Rec. 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

MEM01-C

# CERT C: Rec. MEM02-C

Immediately cast the result of a memory allocation function call into a pointer to the allocated type

## Description

### Rule Definition

*Immediately cast the result of a memory allocation function call into a pointer to the allocated type.*

### Polyspace Implementation

The rule checker checks for **Wrong allocated object size for cast**.

## Examples

### Wrong allocated object size for cast

#### Issue

**Wrong allocated object size for cast** occurs during pointer conversion when the pointer's address is misaligned. If a pointer is converted to a different pointer type, the size of the allocated memory must be a multiple of the size of the destination pointer.

#### Risk

Dereferencing a misaligned pointer has undefined behavior and can cause your program to crash.

#### Fix

Suppose you convert a pointer `ptr1` to `ptr2`. If `ptr1` points to a buffer of `N` bytes and `ptr2` is a *`type`* * pointer where `sizeof(`*`type`*`)` is `n` bytes, make sure that `N` is an integer multiple of `n`.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Dynamic Allocation of Pointers

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(13);
    long *dest;

    dest = (long*)ptr; //defect //Noncompliant
}
```

In this example, the software raises a defect on the conversion of `ptr` to a `long*`. The dynamically allocated memory of `ptr`, 13 bytes, is not a multiple of the size of `dest`, 4 bytes. This misalignment causes the **Wrong allocated object size for cast** defect.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the allocated memory to 12 instead of 13.

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(12);
    long *dest;

    dest = (long*)ptr;
}
```

#### Example - Static Allocation of Pointers

```
void static_non_align(void){
    char arr[13], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr; //defect //Noncompliant
}
```

In this example, the software raises a defect on the conversion of `ptr` to an `int*` in line 6. `ptr` has a memory size of 13 bytes because the array `arr` has a size of 13 bytes. The size of `dest` is 4 bytes, which is not a multiple of 13. This misalignment causes the **Wrong allocated object size for cast** defect.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the size of the array `arr` to a multiple of 4.

```
void static_non_align(void){
    char arr[12], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr;
}
```

#### Example - Allocation with a Function

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;
    dest1 = (int*)my_alloc(13);  //defect //Noncompliant
    dest2 = (char*)my_alloc(13); //not a defect
}
```

In this example, the software raises a defect on the conversion of the pointer returned by `my_alloc(13)` to an `int*` in line 11. `my_alloc(13)` returns a pointer with a dynamically allocated size of 13 bytes. The size of `dest1` is 4 bytes, which is not a divisor of 13. This misalignment causes the **Wrong allocated object size for cast** defect. In line 12, the same function call, `my_alloc(13)`, does not call a defect for the conversion to `dest2` because the size of `char*`, 1 byte, a divisor of 13.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the argument for `my_alloc` to a multiple of 4.

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(12);
    dest2 = (char*)my_alloc(13);
}
```

## Check Information

**Group:** Rec. 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MEM02-C

# CERT C: Rec. MEM03-C

Clear sensitive information stored in reusable resources

## Description

### Rule Definition

*Clear sensitive information stored in reusable resources.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Sensitive heap memory not cleared before release**.
- **Uncleared sensitive data in stack**.

## Examples

### Sensitive heap memory not cleared before release

#### Issue

**Sensitive heap memory not cleared before release** detects dynamically allocated memory containing sensitive data. If you do not clear the sensitive data when you free the memory, Bug Finder raises a defect on the `free` function.

#### Risk

If the memory zone is reallocated, an attacker can still inspect the sensitive data in the old memory zone.

#### Fix

Before calling `free`, clear out the sensitive data using `memset` or `SecureZeroMemory`.

#### Example - Sensitive Buffer Freed, Not Cleared

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    free(buf); //Noncompliant
}
```

In this example, the function uses a buffer of passwords and frees the memory before the end of the function. However, the data in the memory is not cleared by using the `free` command.

#### Correction — Nullify Data

One possible correction is to write over the data to clear out the sensitive information. This example uses `memset` to write over the data with zeros.

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0;i<(sizeof(arr)/sizeof(arr[0]));i++) assert(arr[i]==0)

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);

    if (buf) {
        getpwnam_r(my_user, &pwd, buf, bufsize, &result);
        memset(buf, 0, (size_t)1024);
        isNull(buf);
        free(buf);
    }
}
```

### Uncleared sensitive data in stack

#### Issue

**Uncleared sensitive data in stack** detects static memory containing sensitive data. If you do not clear the sensitive data from your stack before exiting the function or program, Bug Finder raises a defect on the last curly brace.

#### Risk

Leaving sensitive information in your stack, such as passwords or user information, allows an attacker additional access to the information after your program has ended.

#### Fix

Before exiting a function or program, clear out the memory zones that contain sensitive data by using `memset` or `SecureZeroMemory`.

#### Example - Static Buffer of Password Information

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>

void bug_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
}  //Noncompliant
```

In this example, a static buffer is filled with password information. The program frees the stack memory at the end of the program. However, the data is still accessible from the memory.

#### Correction — Clear Memory

One possible correction is to write over the memory before exiting the function. This example uses `memset` to clear the data from the buffer memory.

```
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0; i<(sizeof(arr)/sizeof(arr[0])); i++) assert(arr[i]==0)

void corrected_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    memset(buf, 0, (size_t)1024);
    isNull(buf);
}
```

## Check Information

**Group:** Rec. 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

MEM03-C

# CERT C: Rec. MEM04-C

Beware of zero-length allocations

## Description

### Rule Definition

*Beware of zero-length allocations.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Zero length memory allocation**
- **Variable length array with nonpositive size**.
- **Tainted size of variable length array**.

### Extend Checker

A default Bug Finder analysis might not flag a **Tainted size of variable length array** issue for certain inputs that originate outside of the current analysis boundary. See "Sources of Tainting in a Polyspace Analysis". To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Zero length memory allocation

#### Issue

**Zero length memory allocation** occurs when a memory allocation function such as `malloc` or `calloc` takes a size argument (or number of elements) that might contain the value zero.

#### Risk

According to the C Standard (C11, Subclause 7.22.3), if zero-sized memory is requested from a memory allocation function, the behavior is implementation-defined. In some implementations, the function might return NULL and your existing guards against NULL might suffice to protect against the zero length allocation. In other cases, the function might return a memory region that must not be accessed. Attempts to dereference this region results in undefined behavior.

#### Fix

Check the size to be passed to `malloc`, or the size and number of elements to be passed to `calloc`, for zero.

#### Example – Possibly Zero Size Argument to `malloc`

```
#include <stdlib.h>

void  func(unsigned int size) {
    int *list = (int *)malloc(size);//Noncompliant
    if (list == NULL) {
      /* Handle allocation error */
    }
    else {
    /* Continue processing list */
    }
}
```

In this example, the variable `size` might contain the value zero leading to a zero length memory allocation.

#### Correction – Check for Zero Before Using Variable as Size

Validate external inputs for zero values before using as size argument to the `malloc` function.

```
#include <stdlib.h>

void  func(unsigned int size) {
    if(size == 0) {
        /* Handle zero size error */
    }
    else {
        int *list = (int *)malloc(size);
        if (list == NULL) {
          /* Handle allocation error */
        }
        else {
          /* Continue processing list */
        }
    }
}
```

### Variable length array with nonpositive size

#### Issue

**Variable length array with non-positive size** occurs when size of a variable-length array is zero or negative.

#### Risk

If the size of a variable-length array is zero or negative, unexpected behavior can occur, such as stack overflow.

#### Fix

When you declare a variable-length array as a local variable in a function:

- If you use a function parameter as the array size, check that the parameter is positive.
- If you use the result of a computation on a function parameter as the array size, check that the result is positive.

You can place a test for positive value either before the function call or the array declaration in the function body.

#### Example - Nonpositive Array Size

```
int input(void);
```

```
void add_scalar(int n, int m) {
    int r=0;
    int arr[m][n]; //Noncompliant
    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            arr[i][j] = input();
            r += arr[i][j];
        }
    }
}

void main() {
    add_scalar(2,2);
    add_scalar(-1,2);
    add_scalar(2,0);
}
```

In this example, the second and third calls to `add_scalar` result in a negative and zero size of `arr`.

#### Correction — Make Array Size Positive

One possible correction is fix or remove calls that result in a nonpositive array size.

### Tainted size of variable length array

#### Issue

**Tainted size of variable length array** detects variable length arrays (VLA) whose size is from an unsecure source.

#### Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

#### Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

#### Example - User Input Used as Size of VLA

```
#include<stdio.h>
#inclule<stdlib.h>
#define LIM 40

long squaredSum(int size) {

    int tabvla[size]; //Noncompliant
    long res = 0;
    for (int i=0 ; i<LIM-1 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

**Correction — Check VLA Size**

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 0 and less than 40, before creating the VLA

```
#include <stdio.h>
#include <stdlib.h>
#define LIM 40

long squaredSum(int size) {
    long res = 0;
    if (size>0 && size<LIM){
        int tabvla[size];
        for (int i=0 ; i<size || i<LIM-1 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }else{
        res = -1;
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

## Check Information

**Group:** Rec. 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MEM04-C

# CERT C: Rec. MEM05-C

Avoid large stack allocations

## Description

### Rule Definition

*Avoid large stack allocations.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Direct or indirect function call to itself**.
- **Variable length array with nonpositive size**.
- **Tainted size of variable length array**.

### Extend Checker

A default Bug Finder analysis might not flag a **Tainted size of variable length array** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Direct or indirect function call to itself

#### Issue

The issue occurs when your code contains functions that call themselves directly or indirectly.

#### Risk

Variables local to a function are stored in the call stack. If a function calls itself directly or indirectly several times, the available stack space can be exceeded, causing serious failure. Unless the recursion is tightly controlled, it is difficult to determine the maximum stack space required.

#### Example - Direct and Indirect Recursion

```
void foo1( void ) {     /* Non-compliant */
          /*- Indirect recursion foo1->foo2->foo1... */
    foo2();
    foo1();   /* Non-compliant - Direct recursion */
}

void foo2( void ) {/*Noncompliant*/
         /* Indirect Recursion - Foo2->foo1->foo2*/
    foo1();
}
```

In this example, the rule is violated because of:

- Direct recursion `foo1` → `foo1`.
- Indirect recursion `foo1` → `foo2` → `foo1`.
- Indirect recursion `foo2` → `foo1` → `foo2`.

### Variable length array with nonpositive size

#### Issue

**Variable length array with non-positive size** occurs when size of a variable-length array is zero or negative.

#### Risk

If the size of a variable-length array is zero or negative, unexpected behavior can occur, such as stack overflow.

#### Fix

When you declare a variable-length array as a local variable in a function:

- If you use a function parameter as the array size, check that the parameter is positive.
- If you use the result of a computation on a function parameter as the array size, check that the result is positive.

You can place a test for positive value either before the function call or the array declaration in the function body.

#### Example - Nonpositive Array Size

```
int input(void);

void add_scalar(int n, int m) {
    int r=0;
    int arr[m][n]; //Noncompliant
    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            arr[i][j] = input();
            r += arr[i][j];
        }
    }
}

void main() {
    add_scalar(2,2);
    add_scalar(-1,2);
    add_scalar(2,0);
}
```

In this example, the second and third calls to `add_scalar` result in a negative and zero size of `arr`.

#### Correction — Make Array Size Positive

One possible correction is fix or remove calls that result in a nonpositive array size.

### Tainted size of variable length array

#### Issue

**Tainted size of variable length array** detects variable length arrays (VLA) whose size is from an unsecure source.

#### Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

#### Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

#### Example — User Input Argument Used as Size of VLA

```
#include<stdio.h>
#inclule<stdlib.h>
#define LIM 40

long squaredSum(int size) {

    int tabvla[size]; //Noncompliant
    long res = 0;
    for (int i=0 ; i<LIM-1 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

#### Correction — Check VLA Size

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 10 and less than 100, before creating the VLA

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
```

```
int taintedvlasize(int size) {
    int res = 0;
    if (size>SIZE10 && size<SIZE100) {
        int tabvla[size];
        for (int i=0 ; i<SIZE10 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }
    return res;
}
```

## Check Information

**Group:** Rec. 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MEM05-C

# CERT C: Rec. MEM06-C

Ensure that sensitive data is not written out to disk

## Description

### Rule Definition

*Ensure that sensitive data is not written out to disk.*

### Polyspace Implementation

The rule checker checks for **Sensitive data printed out**.

## Examples

### Sensitive data printed out

#### Issue

**Sensitive data printed out** detects print functions, such as `stdout` or `stderr`, that print sensitive information.

The checker considers the following as sensitive information:

- Return values of password manipulation functions such as `getpw`, `getpwnam` or `getpwuid`.
- Input values of functions such as the Windows-specific function `LogonUser`.

#### Risk

Printing sensitive information, such as passwords or user information, allows an attacker additional access to the information.

#### Fix

One fix for this defect is to not print out sensitive information.

If you are saving your logfile to an external file, set the file permissions so that attackers cannot access the logfile information.

#### Example - Printing Passwords

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

extern void verify_null(const char* buf);
void bug_sensitivedataprint(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    puts("Name\n");
    puts(pwd.pw_name);
    puts("PassWord\n");
    puts(pwd.pw_passwd); //Noncompliant
    memset(buf, 0, sizeof(buf));
    verify_null(buf);
}
```

In this example, Bug Finder flags `puts` for printing out the password `pwd.pw_passwd`.

#### Correction — Obfuscate the Password

One possible correction is to obfuscate the password information so that the information is not visible.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

extern void verify_null(const char* buf);

void sensitivedataprint(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    puts("Name\n");
    puts(pwd.pw_name);
    puts("PassWord\n");
    puts("XXXXXXXX\n");
    memset(buf, 0, sizeof(buf));
    verify_null(buf);
}
```

## Check Information

**Group:** Rec. 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

MEM06-C

# CERT C: Rec. MEM11-C

Do not assume infinite heap space

## Description

### Rule Definition

*Do not assume infinite heap space.*

### Polyspace Implementation

The rule checker checks for **Unprotected dynamic memory allocation**.

## Examples

### Unprotected dynamic memory allocation

#### Issue

**Unprotected dynamic memory allocation** occurs when you do not check after dynamic memory allocation whether the memory allocation succeeded.

#### Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

#### Fix

Check the return value of `malloc`, `calloc`, or `realloc` for NULL before accessing the allocated memory location.

```
int *ptr = malloc(size * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

#### Example - Unprotected dynamic memory allocation error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));

  *p = 2;   //Noncompliant
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function `calloc` returns `NULL` to `p`. Before accessing the memory through `p`, the code does not check whether `p` is `NULL`

#### Correction — Check for NULL Value

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

## Check Information

**Group:** Rec. 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MEM11-C

# CERT C: Rec. MEM12-C

Consider using a goto chain when leaving a function on error when using and releasing resources

## Description

### Rule Definition

*Consider using a goto chain when leaving a function on error when using and releasing resources.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Memory leak**.
- **Resource leak**.

## Examples

### Memory leak

#### Issue

**Memory leak** occurs when you do not free a block of memory allocated through `malloc`, `calloc`, `realloc`, or `new`. If the memory is allocated in a function, the defect does not occur if:

- Within the function, you free the memory using `free` or `delete`.
- The function returns the pointer assigned by `malloc`, `calloc`, `realloc`, or `new`.
- The function stores the pointer in a global variable or in a parameter.

#### Risk

Dynamic memory allocation functions such as `malloc` allocate memory on the heap. If you do not release the memory after use, you reduce the amount of memory available for another allocation. On embedded systems with limited memory, you might end up exhausting available heap memory even during program execution.

#### Fix

Determine the scope where the dynamically allocated memory is accessed. Free the memory block at the end of this scope.

To free a block of memory, use the `free` function on the pointer that was used during memory allocation. For instance:

```
ptr = (int*)malloc(sizeof(int));
//...
free(ptr);
```

It is a good practice to allocate and free memory in the same module at the same level of abstraction. For instance, in this example, `func` allocates and frees memory at the same level but `func2` does not.

```
void func() {
  ptr = (int*)malloc(sizeof(int));
  {
    .//..
  }
  free(ptr);
}

void func2() {
  {
   ptr = (int*)malloc(sizeof(int));
   //...
  }
  free(ptr);
}
```

See CERT-C Rule MEM00-C.

#### Example - Dynamic Memory Not Released Before End of Function

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
} //Noncompliant
```

In this example, `pi` is dynamically allocated by `malloc`. The function `assign_memory` does not free the memory, nor does it return `pi`.

#### Correction — Free Memory

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
    free(pi);
}
```

#### Correction — Return Pointer from Dynamic Allocation

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

```
#include<stdlib.h>
#include<stdio.h>

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
            printf("Memory allocation failed");
            return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

### Resource leak

#### Issue

**Resource leak** occurs when you open a file stream by using a `FILE` pointer but do not close it before:

- The end of the pointer’s scope.
- Assigning the pointer to another stream.

#### Risk

If you do not release file handles explicitly as soon as possible, a failure can occur due to exhaustion of resources.

#### Fix

Close a `FILE` pointer before the end of its scope, or before you assign the pointer to another stream.

#### Example - FILE Pointer Not Released Before End of Scope

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" ); //Noncompliant
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`.

#### Correction — Release FILE Pointer

One possible correction is to explicitly dissociate `fp1` from the file stream of `data1.txt`.

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );
    fclose(fp1);

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

## Check Information

**Group:** Rec. 08. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

MEM12-C

# CERT C: Rec. FIO02-C

Canonicalize path names originating from tainted sources

## Description

### Rule Definition

*Canonicalize path names originating from tainted sources.*

### Polyspace Implementation

The rule checker checks for **Vulnerable path manipulation**.

## Examples

### Vulnerable path manipulation

#### Issue

**Vulnerable path manipulation** detects relative or absolute path traversals. If the path traversal contains a tainted source, or you use the path to open/create files, Bug Finder raises a defect.

#### Risk

Relative path elements, such as `".."` can resolve to locations outside the intended folder. Absolute path elements, such as `"/abs/path"` can also resolve to locations outside the intended folder.

An attacker can use these types of path traversal elements to traverse to the rest of the file system and access other files or folders.

#### Fix

Avoid vulnerable path traversal elements such as `/../` and `/abs/path/`. Use fixed file names and locations wherever possible.

#### Example - Relative Path Traversal

```
# include <stdio.h>
# include <string.h>
# include <wchar.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <unistd.h>
# include <stdlib.h>
# define BASEPATH "/tmp/"
# define FILENAME_MAX 512

static void Relative_Path_Traversal(void)
{
    char * data;
    char data_buf[FILENAME_MAX] = BASEPATH;
    char sub_buf[FILENAME_MAX];
    if (fgets(sub_buf, FILENAME_MAX, stdin) == NULL) exit (1);
    data = data_buf;
    strcat(data, sub_buf);

    FILE *file = NULL;
    file = fopen(data, "wb+");  //Noncompliant
    if (file != NULL) fclose(file);
}

int path_call(void){
    Relative_Path_Traversal();
}
```

This example opens a file from `"/tmp/"`, but uses a relative path to the file. An external user can manipulate this relative path when `fopen` opens the file.

#### Correction — Use Fixed File Name

One possible correction is to use a fixed file name instead of a relative path. This example uses `file.txt`.

```
# include <stdio.h>
# include <string.h>
# include <wchar.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <unistd.h>
# include <stdlib.h>
# define BASEPATH "/tmp/"
# define FILENAME_MAX 512

static void Relative_Path_Traversal(void)
{
    char * data;
    char data_buf[FILENAME_MAX] = BASEPATH;
    data = data_buf;

    /* FIX: Use a fixed file name */
    strcat(data, "file.txt");
    FILE *file = NULL;
    file = fopen(data, "wb+");
    if (file != NULL) fclose(file);
}

int path_call(void){
    Relative_Path_Traversal();
}
```

## Check Information

**Group:** Rec. 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO02-C

# CERT C: Rec. FIO11-C

Take care when specifying the mode parameter of fopen()

## Description

### Rule Definition

*Take care when specifying the mode parameter of fopen().*

### Polyspace Implementation

The rule checker checks for **Bad file access mode or status**.

## Examples

### Bad file access mode or status

#### Issue

**Bad file access mode or status** occurs when you use functions in the `fopen` or `open` group with invalid or incompatible file access modes, file creation flags, or file status flags as arguments. For instance, for the `open` function, examples of valid:

- Access modes include `O_RDONLY`, `O_WRONLY`, and `O_RDWR`
- File creation flags include `O_CREAT`, `O_EXCL`, `O_NOCTTY`, and `O_TRUNC`.
- File status flags include `O_APPEND`, `O_ASYNC`, `O_CLOEXEC`, `O_DIRECT`, `O_DIRECTORY`, `O_LARGEFILE`, `O_NOATIME`, `O_NOFOLLOW`, `O_NONBLOCK`, `O_NDELAY`, `O_SHLOCK`, `O_EXLOCK`, `O_FSYNC`, `O_SYNC` and so on.

The defect can occur in the following situations.

| Situation | Risk | Fix |
|---|---|---|
| You pass an empty or invalid access mode to the `fopen` function.<br>According to the ANSI C standard, the valid access modes for `fopen` are:<br>• `r`,`r+`<br>• `w`,`w+`<br>• `a`,`a+`<br>• `rb`, `wb`, `ab`<br>• `r+b`, `w+b`, `a+b`<br>• `rb+`, `wb+`, `ab+` | `fopen` has undefined behavior for invalid access modes.<br>Some implementations allow extension of the access mode such as:<br>• GNU: `rb+cmxe,ccs=utf`<br>• Visual C++: `a+t`, where `t` specifies a text mode.<br>However, your access mode string must begin with one of the valid sequences. | Pass a valid access mode to `fopen`. |

| Situation | Risk | Fix |
|---|---|---|
| You pass the status flag O_APPEND to the open function without combining it with either O_WRONLY or O_RDWR. | O_APPEND indicates that you intend to add new content at the end of a file. However, without O_WRONLY or O_RDWR, you cannot write to the file.<br>The open function does not return -1 for this logical error. | Pass either O_APPEND\|O_WRONLY or O_APPEND\|O_RDWR as access mode. |
| You pass the status flags O_APPEND and O_TRUNC together to the open function. | O_APPEND indicates that you intend to add new content at the end of a file. However, O_TRUNC indicates that you intend to truncate the file to zero. Therefore, the two modes cannot operate together.<br>The open function does not return -1 for this logical error. | Depending on what you intend to do, pass one of the two modes. |
| You pass the status flag O_ASYNC to the open function. | On certain implementations, the mode O_ASYNC does not enable signal-driven I/O operations. | Use the fcntl(pathname, F_SETFL, O_ASYNC); instead. |

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface".

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Invalid Access Mode with fopen

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "rw"); //Noncompliant
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

In this example, the access mode rw is invalid. Because r indicates that you open the file for reading and w indicates that you create a new file for writing, the two access modes are incompatible.

### Correction — Use Either `r` or `w` as Access Mode

One possible correction is to use the access mode corresponding to what you intend to do.

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "w");
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

## Check Information

**Group:** Rec. 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO11-C

# CERT C: Rec. FIO21-C

Do not create temporary files in shared directories

## Description

### Rule Definition

*Do not create temporary files in shared directories.*

### Polyspace Implementation

The rule checker checks for **Use of non-secure temporary file**.

## Examples

### Use of non-secure temporary file

#### Issue

**Use of non-secure temporary file** looks for temporary file routines that are not secure.

#### Risk

If an attacker guesses the file name generated by a standard temporary file routine, the attacker can:

- Cause a race condition when you generate the file name.
- Precreate a file of the same name, filled with malicious content. If your program reads the file, the attacker's file can inject the malicious code.
- Create a symbolic link to a file storing sensitive data. When your program writes to the temporary file, the sensitive data is deleted.

#### Fix

To create temporary files, use a more secure standard temporary file routine, such as `mkstemp` from POSIX.1-2001.

Also, when creating temporary files with routines that allow flags, such as `mkostemp`, use the exclusion flag `O_EXCL` to avoid race conditions.

#### Example - Temp File Created With `tempnam`

```
#define _BSD_SOURCE
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int test_temp()
```

```
{
    char tpl[] = "abcXXXXXX";
    char suff_tpl[] = "abcXXXXXXsuff";
    char *filename = NULL;
    int fd;

    filename = tempnam("/var/tmp", "foo_");

    if (filename != NULL)
    {
        printf("generated tmp name (%s) in (%s:%s:%s)\n",
               filename, getenv("TMPDIR") ? getenv("TMPDIR") : "$TMPDIR",
               "/var/tmp", P_tmpdir);

        fd = open(filename, O_CREAT, S_IRWXU|S_IRUSR); //Noncompliant
        if (fd != -1)
        {
            close(fd);
            unlink(filename);
            return 1;
        }
    }
    return 0;
}
```

In this example, Bug Finder flags open because it tries to use an unsecure temporary file. The file is opened without exclusive privileges. An attacker can access the file causing various risks on page 13-84.

**Correction — Add O_EXCL Flag**

One possible correction is to add the O_EXCL flag when you open the temporary file.

```
#define _BSD_SOURCE
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int test_temp()
{
    char tpl[] = "abcXXXXXX";
    char suff_tpl[] = "abcXXXXXXsuff";
    char *filename = NULL;
    int fd;

    filename = tempnam("/var/tmp", "foo_");

    if (filename != NULL)
    {
        printf("generated tmp name (%s) in (%s:%s:%s)\n",
               filename, getenv("TMPDIR") ? getenv("TMPDIR") : "$TMPDIR",
               "/var/tmp", P_tmpdir);
        fd = open(filename, O_CREAT|O_EXCL, S_IRWXU|S_IRUSR);
        if (fd != -1)
        {
            close(fd);
            unlink(filename);
            return 1;
        }
    }
    return 0;
}
```

## Check Information

**Group:** Rec. 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO21-C

# CERT C: Rec. FIO24-C

Do not open a file that is already open

## Description

### Rule Definition

*Do not open a file that is already open.*

### Polyspace Implementation

The rule checker checks for **Opening previously opened resource**.

## Examples

### Opening previously opened resource

#### Issue

**Opening previously opened resource** checks for file opening functions that are opening an already opened file.

#### Risk

If you open a resource multiple times, you can encounter:

- A race condition when accessing the file.
- Undefined or unexpected behavior for that file.
- Portability issues when you run your program on different targets.

#### Fix

Once a resource is open, close the resource before reopening.

#### Example - File Reopened With New Permissions

```
#include <stdio.h>
const char* logfile = "my_file.log";

void doubleresourceopen()
{
    FILE* fpa = fopen(logfile, "w");
    if (fpa == NULL) {
        return;
    }
    (void)fprintf(fpa, "Writing");
    FILE* fpb = fopen(logfile, "r"); //Noncompliant
    (void)fclose(fpa);
    (void)fclose(fpb);
}
```

In this example, a `logfile` is opened in the first line of this function with write privileges. Halfway through the function, the `logfile` is opened again with read privileges.

#### Correction — Close Before Reopening

One possible correction is to close the file before reopening the file with different privileges.

```
#include <stdio.h>
const char* logfile = "my_file.log";

void doubleresourceopen()
{
    FILE* fpa = fopen(logfile, "w");
    if (fpa == NULL) {
        return;
    }
    (void)fprintf(fpa, "Writing");
    (void)fclose(fpa);
    FILE* fpb = fopen(logfile, "r");
    (void)fclose(fpb);
}
```

## Check Information

**Group:** Rec. 09. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO24-C

# CERT C: Rec. ENV01-C

Do not make assumptions about the size of an environment variable

## Description

### Rule Definition

*Do not make assumptions about the size of an environment variable.*

### Polyspace Implementation

The rule checker checks for **Tainted NULL or non-null-terminated string**.

### Extend Checker

A default Bug Finder analysis might not flag a **Tainted NULL or non-null-terminated string** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Tainted NULL or non-null-terminated string

#### Issue

**Tainted NULL or non-null-terminated string** looks for strings from unsecure sources that are being used in string manipulation routines that implicitly dereference the string buffer. For example, `strcpy` or `sprintf`.

**Tainted NULL or non-null-terminated string** raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

**Note** If you reference a string using the form `ptr[i]`, `*ptr`, or pointer arithmetic, Bug Finder raises a **Use of tainted pointer** defect instead. The **Tainted NULL or non-null-terminated string** defect is raised only when the pointer is used as a string.

#### Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

#### Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

**Example - Getting String from Input Argument**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1)); //Noncompliant
    print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

**Correction — Validate the Data**

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // TAINTED_STRING only flagged here //Noncompliant
        // - string is not null
        // - string has a positive and limited size
        // - TAINTED_STRING on strlen used as a firewall
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

#### Correction — Validate the Data

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

## Check Information

**Group:** Rec. 10. Environment (ENV)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

**External Websites**
ENV01-C

# CERT C: Rec. ERR00-C

Adopt and implement a consistent and comprehensive error-handling policy

## Description

### Rule Definition

*Adopt and implement a consistent and comprehensive error-handling policy.*

### Polyspace Implementation

The rule checker checks for **Returned value of a sensitive function not checked**.

## Examples

### Returned value of a sensitive function not checked

#### Issue

This issue occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

#### Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

#### Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

#### Example – Sensitive Function Return Ignored

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);  //Noncompliant
}
```

This example shows a call to the sensitive function `pthread_attr_init`. The return value of `pthread_attr_init` is ignored, causing a defect.

#### Correction 1 – Cast Function to (void)

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);
}
```

#### Correction 2 – Test Return Value

One possible correction is to test the return value of `pthread_attr_init` to check for errors.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

void initialize() {
    pthread_attr_t attr;
    int result;

    result = pthread_attr_init(&attr);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

#### Example – Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0)); //Noncompliant
    pthread_join(thread_id,  &res);  //Noncompliant
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

**Correction — Test the Return Value of Critical Functions**

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

## Check Information

**Group:** Rec. 12. Error Handling (ERR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR00-C

# CERT C: Rec. API04-C

Provide a consistent and usable error-checking mechanism

## Description

### Rule Definition

*Provide a consistent and usable error-checking mechanism.*

### Polyspace Implementation

The rule checker checks for **Returned value of a sensitive function not checked**.

## Examples

### Returned value of a sensitive function not checked

#### Issue

This issue occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

#### Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

#### Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

#### Example – Sensitive Function Return Ignored

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);  //Noncompliant
}
```

This example shows a call to the sensitive function `pthread_attr_init`. The return value of `pthread_attr_init` is ignored, causing a defect.

#### Correction 1 – Cast Function to (void)

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);
}
```

#### Correction 2 – Test Return Value

One possible correction is to test the return value of `pthread_attr_init` to check for errors.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

void initialize() {
    pthread_attr_t attr;
    int result;

    result = pthread_attr_init(&attr);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

#### Example – Critical Function Return Ignored

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0)); //Noncompliant
    pthread_join(thread_id,  &res);  //Noncompliant
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

#### Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

## Check Information

**Group:** Rec. 13. Application Programming Interfaces (API)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

API04-C

# CERT C: Rec. CON01-C

Acquire and release synchronization primitives in the same module, at the same level of abstraction

## Description

### Rule Definition

*Acquire and release synchronization primitives in the same module, at the same level of abstraction.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Missing lock**.
- **Missing unlock**.
- **Double lock**.
- **Double unlock**.

### Extend Checker

For the rule checker to detect issues, your code must use one of the concurrency primitives recognized by Polyspace for thread creation and shared variable protection. Otherwise, you must explicitly specify tasks, interrupts, critical sections, and other multitasking options in your project configuration. See “Multitasking”.

You can also extend this checker to detect synchronization issues by mapping your multithreading functions to their known POSIX equivalents. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.

## Examples

### Missing lock

#### Issue

**Missing lock** occurs when a task calls an unlock function before calling the corresponding lock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait till `my_task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

#### Risk

A call to an unlock function without a corresponding lock function can indicate a coding error. For instance, perhaps the unlock function does not correspond to the lock function that begins the critical section.

#### Fix

The fix depends on the root cause of the defect. For instance, if the defect occurs because of a mismatch between lock and unlock function, check the lock-unlock function pair in your Polyspace analysis configuration and fix the mismatch.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    // ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   // ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Missing lock

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset(void)
{
  begin_critical_section();
  global_var = 0;
  end_critical_section();
}

void my_task(void)
{
  global_var += 1;
  end_critical_section(); //Noncompliant
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ | |
| **Tasks** on page 2-121 | `my_task`, `reset` | |
| **Critical section details** on page 2-133 | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`. `my_task` calls `end_critical_section` before calling `begin_critical_section`.

#### Correction — Provide Lock

One possible correction is to call the lock function `begin_critical_section` before the instructions in the critical section.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset(void)
{
  begin_critical_section();
  global_var = 0;
  end_critical_section();
}

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
    end_critical_section();
}
```

#### Example - Lock in Condition

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;
```

```
void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      if(index%10==0) {
        begin_critical_section();
        global_var ++;
      }
      end_critical_section();  //Noncompliant
      index++;
    }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ | |
| **Tasks** on page 2-121 | `my_task`, `reset` | |
| **Critical section details** on page 2-133 | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`.

In the `while` loop, `my_task` leaves a critical section through the call `end_critical_section();`. In an iteration of the `while` loop:

- If `my_task` enters the `if` condition branch, the critical section begins through a call to `begin_critical_section`.
- If `my_task` does not enter the `if` condition branch and leaves the `while` loop, the critical section does not begin. Therefore, a **Missing lock** defect occurs.
- If `my_task` does not enter the `if` condition branch and continues to the next iteration of the `while` loop, the unlock function `end_critical_section` is called again. A **Double unlock** defect occurs.

Because `numCycles` is a `volatile` variable, it can take any value. Any of the cases above are possible. Therefore, a **Missing lock** defect and a **Double unlock** defect appear on the call `end_critical_section`.

### Missing unlock

#### Issue

**Missing unlock** occurs when:

- A task calls a lock function.
- The task ends without a call to an unlock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task, `my_task`, calls a lock function, `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, before analysis, you must specify the multitasking options. On the **Configuration** pane, select **Multitasking**.

#### Risk

An unlock function ends a critical section so that other waiting tasks can enter the critical section. A missing unlock function can result in tasks blocked for an unnecessary length of time.

#### Fix

Identify the critical section of code, that is, the section that you want to be executed as an atomic block. At the end of this section, call the unlock function that corresponds to the lock function used at the beginning of the section.

There can be other reasons and corresponding fixes for the defect. Perhaps you called the incorrect unlock function. Check the lock-unlock function pair in your Polyspace analysis configuration and fix the mismatch.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    // ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   // ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.

- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example - Missing Unlock**

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset()
{
    begin_critical_section();
    global_var = 0;
    end_critical_section();
}

void my_task(void)
{
    begin_critical_section();  //Noncompliant
    global_var += 1;
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ | |
| **Tasks** on page 2-121 | `my_task`, `reset` | |
| **Critical section details** on page 2-133 | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`. `my_task` enters a critical section through the call `begin_critical_section();`. `my_task` ends without calling `end_critical_section`.

**Correction — Provide Unlock**

One possible correction is to call the unlock function `end_critical_section` after the instructions in the critical section.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset(void)
{
    begin_critical_section();
    global_var = 0;
    end_critical_section();
}

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
    end_critical_section();
}
```

#### Example - Unlock in Condition

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section(); //Noncompliant
      global_var ++;
      if(index%10==0) {
        global_var = 0;
        end_critical_section();
      }
      index++;
    }
}
```

In this example, to emulate multitasking behavior, specify the following options.

| Option | Specification |
|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ |
| **Tasks** on page 2-121 | `my_task`, `reset` |

| Option | Specification | |
|---|---|---|
| **Critical section details** on page 2-133 | **Starting routine** | **Ending routine** |
| | begin_critical_section | end_critical_section |

On the command-line, you can use the following:

```
polyspace-bug-finder
   -entry-points my_task,reset
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

The example has two entry points, `my_task` and `reset`.

In the `while` loop, `my_task` enters a critical section through the call `begin_critical_section();`. In an iteration of the `while` loop:

- If `my_task` enters the `if` condition branch, the critical section ends through a call to `end_critical_section`.
- If `my_task` does not enter the `if` condition branch and leaves the `while` loop, the critical section does not end. Therefore, a **Missing unlock** defect occurs.
- If `my_task` does not enter the `if` condition branch and continues to the next iteration of the `while` loop, the lock function `begin_critical_section` is called again. A **Double lock** defect occurs.

Because `numCycles` is a `volatile` variable, it can take any value. Any of the cases above is possible. Therefore, a **Missing unlock** defect and a **Double lock** defect appear on the call `begin_critical_section`.

#### Correction — Place Unlock Outside Condition

One possible correction is to call the unlock function `end_critical_section` outside the `if` condition.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        global_var=0;
      }
      end_critical_section();
      index++;
    }
}
```

#### Correction — Place Unlock in Every Conditional Branch

Another possible correction is to call the unlock function `end_critical_section` in every branches of the `if` condition.

```
void begin_critical_section(void);
void end_critical_section(void);

int global_var;

void reset() {
    begin_critical_section();
    global_var=0;
    end_critical_section();
}

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        global_var=0;
        end_critical_section();
      }
      else
        end_critical_section();
      index++;
    }
}
```

### Double lock

#### Issue

**Double lock** occurs when:

- A task calls a lock function `my_lock`.
- The task calls `my_lock` again before calling the corresponding unlock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `task1` calls a lock function `lock`, other tasks calling `lock` must wait until `task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

**Risk**

A call to a lock function begins a critical section so that other tasks have to wait to enter the same critical section. If the same lock function is called again within the critical section, the task blocks itself.

**Fix**

The fix depends on the root cause of the defect. A double lock defect often indicates a coding error. Perhaps you omitted the call to an unlock function to end a previous critical section and started the next critical section. Perhaps you wanted to use a different lock function for the second critical section.

Identify each critical section of code, that is, the section that you want to be executed as an atomic block. Call a lock function at the beginning of the section. Within the critical section, make sure that you do not call the lock function again. At the end of the section, call the unlock function that corresponds to the lock function.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    // ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   // ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Double Lock**

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    lock();
    global_var += 1;
    lock();  //Noncompliant
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ | |
| **Tasks** on page 2-121 | `my_task`, `reset` | |
| **Critical section details** on page 2-133 | **Starting routine** | **Ending routine** |
| | `lock` | `unlock` |

On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2
   -critical-section-begin lock:cs1
   -critical-section-end unlock:cs1
```

`task1` enters a critical section through the call `lock();`. `task1` calls `lock` again before it leaves the critical section through the call `unlock();`.

#### Correction — Remove First Lock

If you want the first `global_var+=1;` to be outside the critical section, one possible correction is to remove the first call to `lock`. However, if other tasks are using `global_var`, this code can produce a `Data race` error.

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    global_var += 1;
    lock();
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

**Correction — Remove Second Lock**

If you want the first `global_var+=1;` to be inside the critical section, one possible correction is to remove the second call to `lock`.

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    lock();
    global_var += 1;
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

**Correction — Add Another Unlock**

If you want the second `global_var+=1;` to be inside a critical section, another possible correction is to add another call to `unlock`.

```
int global_var;

void lock(void);
void unlock(void);

void task1(void)
{
    lock();
    global_var += 1;
    unlock();
    lock();
    global_var += 1;
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

**Example - Double Lock with Function Call**

```
int global_var;

void lock(void);
void unlock(void);

void performOperation(void) {
  lock(); //Noncompliant
  global_var++;
}

void task1(void)
{
    lock();
    global_var += 1;
    performOperation();
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ | |
| **Tasks** on page 2-121 | `my_task`, `reset` | |
| **Critical section details** on page 2-133 | **Starting routine** | **Ending routine** |
| | `lock` | `unlock` |

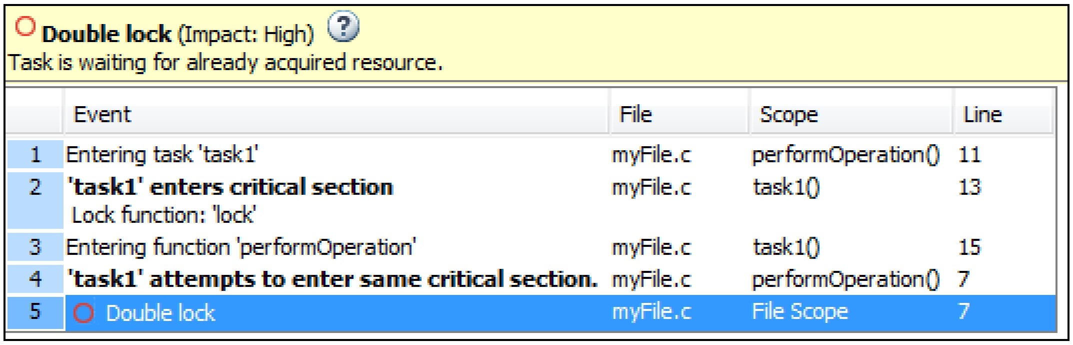
On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2
   -critical-section-begin lock:cs1
   -critical-section-end unlock:cs1
```

`task1` enters a critical section through the call `lock();`. `task1` calls the function `performOperation`. In `performOperation`, `lock` is called again even though `task1` has not left the critical section through the call `unlock();`.

In the result details for the defect, you see the sequence of instructions leading to the defect. For instance, you see that following the first entry into the critical section, the execution path:

- Enters function `performOperation`.
- Inside `performOperation`, attempts to enter the same critical section once again.



○ **Double lock** (Impact: High)

Task is waiting for already acquired resource.

| | Event | File | Scope | Line |
|---|---|---|---|---|
| 1 | Entering task 'task1' | myFile.c | performOperation() | 11 |
| 2 | **'task1' enters critical section**<br>Lock function: 'lock' | myFile.c | task1() | 13 |
| 3 | Entering function 'performOperation' | myFile.c | task1() | 15 |
| 4 | **'task1' attempts to enter same critical section.** | myFile.c | performOperation() | 7 |
| 5 | ○ Double lock | myFile.c | File Scope | 7 |

You can click each event to navigate to the corresponding line in the source code.

**Correction — Remove Second Lock**

One possible correction is to remove the call to `lock` in `task1`.

```
int global_var;

void lock(void);
void unlock(void);

void performOperation(void) {
  global_var++;
}

void task1(void)
{
    lock();
    global_var += 1;
    performOperation();
    unlock();
}

void task2(void)
{
    lock();
    global_var += 1;
    unlock();
}
```

### Double unlock

#### Issue

**Double unlock** occurs when:

- A task calls a lock function `my_lock`.
- The task calls the corresponding unlock function `my_unlock`.
- The task calls `my_unlock` again. The task does not call `my_lock` a second time between the two calls to `my_unlock`.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `task1` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `task1` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

#### Risk

A double unlock defect can indicate a coding error. Perhaps you wanted to call a different unlock function to end a different critical section. Perhaps you called the unlock function prematurely the first time and only the second call indicates the end of the critical section.

#### Fix

The fix depends on the root cause of the defect.

Identify each critical section of code, that is, the section that you want to be executed as an atomic block. Call a lock function at the beginning of the section. Only at the end of the section, call the unlock function that corresponds to the lock function. Remove any other redundant call to the unlock function.

See examples of fixes below. To avoid the issue, you can follow the practice of calling the lock and unlock functions in the same module at the same level of abstraction. For instance, in this example, `func` calls the lock and unlock function at the same level but `func2` does not.

```
void func() {
  my_lock();
  {
    // ...
  }
  my_unlock();
}

void func2() {
  {
   my_lock();
   // ...
  }
  my_unlock();
}
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Double Unlock**

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION(); //Noncompliant
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Value | |
|---|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ | |
| **Tasks** on page 2-121 | `task1`<br>`task2` | |
| **Critical section details** on page 2-133 | **Starting routine** | **Ending routine** |
| | `BEGIN_CRITICAL_SECTION` | `END_CRITICAL_SECTION` |

On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2
   -critical-section-begin BEGIN_CRITICAL_SECTION:cs1
   -critical-section-end END_CRITICAL_SECTION:cs1
```

`task1` enters a critical section through the call `BEGIN_CRITICAL_SECTION();`. `task1` leaves the critical section through the call `END_CRITICAL_SECTION();`. `task1` calls `END_CRITICAL_SECTION` again without an intermediate call to `BEGIN_CRITICAL_SECTION`.

#### Correction — Remove Second Unlock

If you want the second `global_var+=1;` to be outside the critical section, one possible correction is to remove the second call to `END_CRITICAL_SECTION`. However, if other tasks are using `global_var`, this code can produce a `Data race` error.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    global_var += 1;
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

#### Correction — Remove First Unlock

If you want the second `global_var+=1;` to be inside the critical section, one possible correction is to remove the first call to `END_CRITICAL_SECTION`.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    global_var += 1;
    END_CRITICAL_SECTION();
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

#### Correction — Add Another Lock

If you want the second `global_var+=1;` to be inside a critical section, another possible correction is to add another call to `BEGIN_CRITICAL_SECTION`.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void task1(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}

void task2(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

## Check Information

**Group:** Rec. 14. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

CON01-C

# CERT C: Rec. CON05-C

Do not perform operations that can block while holding a lock

## Description

### Rule Definition

*Do not perform operations that can block while holding a lock.*

### Polyspace Implementation

The rule checker checks for **Blocking operation while holding lock**.

## Examples

### Blocking operation while holding lock

#### Issue

**Blocking operation while holding lock** occurs when a task (thread) performs a potentially lengthy operation while holding a lock.

The checker considers calls to these functions as potentially lengthy:

- Functions that access a network such as `recv`
- System call functions such as `fork`, `pipe` and `system`
- Functions for I/O operations such as `getchar` and `scanf`
- File handling functions such as `fopen`, `remove` and `lstat`
- Directory manipulation functions such as `mkdir` and `rmdir`

The checker automatically detects certain primitives that hold and release a lock, for instance, `pthread_mutex_lock` and `pthread_mutex_unlock`. For the full list of primitives that are automatically detected, see “Auto-Detection of Thread Creation and Critical Section in Polyspace”.

#### Risk

If a thread performs a lengthy operation when holding a lock, other threads that use the lock have to wait for the lock to be available. As a result, system performance can slow down or deadlocks can occur.

#### Fix

Perform the blocking operation before holding the lock or after releasing the lock.

Some functions detected by this checker can be called in a way that does not make them potentially lengthy. For instance, the function `recv` can be called with the parameter `O_NONBLOCK` which causes the call to fail if no message is available. When called with this parameter, `recv` does not wait for a message to become available.

#### Example - Network I/O Operations with `recv` While Holding Lock

```
#include <pthread.h>
#include <sys/socket.h>
```

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void thread_foo(void *ptr) {
  unsigned int num;
  int result;
  int sock;

  /* sock is a connected TCP socket */

  if ((result = pthread_mutex_lock(&mutex)) != 0) {
    /* Handle Error */
  }

  if ((result = recv(sock, (void *)&num, sizeof(unsigned int), 0)) < 0) { //Noncompliant
    /* Handle Error */
  }

  /* ... */

  if ((result = pthread_mutex_unlock(&mutex)) != 0) {
    /* Handle Error */
  }
}

int main() {
  pthread_t thread;
  int result;

  if ((result = pthread_mutexattr_settype(
      &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_init(&mutex, &attr)) != 0) {
    /* Handle Error */
  }

  if (pthread_create(&thread, NULL,(void*(*)(void*))& thread_foo, NULL) != 0) {
    /* Handle Error */
  }

  /* ... */

  pthread_join(thread, NULL);

  if ((result = pthread_mutex_destroy(&mutex)) != 0) {
    /* Handle Error */
  }

  return 0;
}
```

In this example, in each thread created with `pthread_create`, the function `thread_foo` performs a network I/O operation with `recv` after acquiring a lock with `pthread_mutex_lock`. Other threads using the same lock variable mutex have to wait for the operation to complete and the lock to become available.

### Correction — Perform Blocking Operation Before Acquiring Lock

One possible correction is to call recv before acquiring the lock.

```
#include <pthread.h>
#include <sys/socket.h>

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void thread_foo(void *ptr) {
  unsigned int num;
  int result;
  int sock;

  /* sock is a connected TCP socket */
  if ((result = recv(sock, (void *)&num, sizeof(unsigned int), 0)) < 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_lock(&mutex)) != 0) {
    /* Handle Error */
  }

    /* ... */

  if ((result = pthread_mutex_unlock(&mutex)) != 0) {
    /* Handle Error */
  }
}

int main() {
  pthread_t thread;
  int result;

  if ((result = pthread_mutexattr_settype(
      &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle Error */
  }

  if ((result = pthread_mutex_init(&mutex, &attr)) != 0) {
    /* Handle Error */
  }

  if (pthread_create(&thread, NULL,(void*(*)(void*))& thread_foo, NULL) != 0) {
    /* Handle Error */
  }

  /* ... */

  pthread_join(thread, NULL);

  if ((result = pthread_mutex_destroy(&mutex)) != 0) {
    /* Handle Error */
  }
  return 0;
}
```

## Check Information

**Group:** Rec. 14. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CON05-C

# CERT C: Rec. MSC01-C

Strive for logical completeness

## Description

### Rule Definition

*Strive for logical completeness.*

### Polyspace Implementation

The rule checker checks for **Missing case for switch condition**.

## Examples

### Missing case for switch condition

#### Issue

**Missing case for switch condition** occurs when the `switch` variable can take values that are not covered by a `case` statement.

**Note** Bug Finder only raises a defect if the switch variable is not full range.

#### Risk

If the `switch` variable takes a value that is not covered by a `case` statement, your program can have unintended behavior.

A switch-statement that makes a security decision is particularly vulnerable when all possible values are not explicitly handled. An attacker can use this situation to deviate the normal execution flow.

#### Fix

It is good practice to use a `default` statement as a catch-all for values that are not covered by a `case` statement. Even if the `switch` variable takes an unintended value, the resulting behavior can be anticipated.

#### Example - Missing Default Condition

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}

int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )  //Noncompliant
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    }

    printf("Welcome!\n");
    return r;
}
```

In this example, the `enum` parameter `User` can take a value `UNKNOWN` that is not covered by a `case` statement.

**Correction — Add a Default Condition**

One possible correction is to add a default condition for possible values that are not covered by a `case` statement.

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}
```

```
int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    break;
    default:
        printf("Invalid login credentials!\n");
    }

    printf("Welcome!\n");
    return r;
}
```

## Check Information

**Group:** Rec. 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC01-C

# CERT C: Rec. MSC04-C

Use comments consistently and in a readable fashion

## Description

### Rule Definition

*Use comments consistently and in a readable fashion.*

### Polyspace Implementation

The rule checker checks for **Use of /* and // within a comment**.

## Examples

### Use of /* and // within a comment

#### Issue

The issue occurs when you use the character sequences `/*` and `//` within a comment.

You cannot annotate this rule in the source code. For information on annotations, see “Annotate Code and Hide Known or Acceptable Results”.

#### Risk

These character sequences in code comments might indicate an unexpected error. For instance:

- If your code contains a `/*` or a `//` in a `/*  */` comment, it typically means that you have inadvertently commented out code.
- If your code contains a `/*` in a `//` comment, it typically means that you have inadvertently uncommented a `/*  */` comment.
- Your code contains nested comments. Such comments are confusing and might lead to developer confusion.

#### Example - /* Used in // Comments

```
int x,y,z,o;


void non_compliant_comments ( void )
{
    x = y //      /* Noncompliant
    + z
    //  */
    ;
    x = y//**/ z //Noncompliant
    +o;
    x = y //*divisor:*/z  //Noncompliant
    +o;

    z++;    //    Compliant with exception: // permitted within a // comment
}

void compliant_comments ( void )
{
    x = y /*      Compliant
    + z
    */
    ;
    x = y / /*divisor:*/ z //Compliant
    +o;
    z++;    //    Compliant with exception: // is permitted within a // comment
}
```

In this example, in the `non_compliant_comments` function, :

- In the first definition of `x`, the `/*` character occurs in what appears to be a `//` comment, violating the rule. Because of the comment structure, the operation that takes place is `x = y + z;`. However, without the two `//`-s, an entirely different operation `x=y;` takes place. It is not clear which operation is intended.
- In the second and third definition of `x`, the presence of the characters `/*` in what appears to be a `//` comment raises violations of this rule. Because of the confusing comment placement, `x` is defined as `y+o` in these statements, instead of `x = y/z + o`, as intended.

Use a comment format that makes your intention clear. For instance, in the `compliant_comments` function:

- In the first definition of x, it is clear that x is defined as `x = y` and the rest of the code is deliberately commented out;
- In the second definition of `x`, it is clear x is defined as `x = y/z+o`. The placement of the comment `/*divisor*/` is clear and does not hinder code understanding.

it is clear that the operation `x=y;` is intended.

## Check Information

**Group:** Rec. 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MSC04-C

# CERT C: Rec. MSC12-C

Detect and remove code that has no effect or is never executed

## Description

### Rule Definition

*Detect and remove code that has no effect or is never executed.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Unreachable code**.
- **Dead code**.
- **Useless if**.
- **Write without a further read**.

## Examples

### Unreachable code

#### Issue

The issue occurs when your project contains code that is unreachable.

#### Risk

Unless a program exhibits any undefined behavior, unreachable code cannot execute. The unreachable code cannot affect the program output. The presence of unreachable code can indicate an error in the program logic. Unreachable code that the compiler does not remove wastes resources, for example:

- It occupies space in the target machine memory.
- Its presence can cause a compiler to select longer, slower jump instructions when transferring control around the unreachable code.
- Within a loop, it can prevent the entire loop from residing in an instruction cache.

#### Example - Code Following `return` Statement

```
enum light { red, amber, red_amber, green };
void report_color(enum light);
enum light next_light ( enum light color )
{
    enum light res;

    switch ( color )
    {
    case red:
        res = red_amber;
        break;
    case red_amber:
        res = green;
        break;
    case green:
        res = amber;
        break;
    case amber:
        res = red;
        break;
    default:
    {
        error_handler ();
        break;
    }
    }

    report_color(res);
    return res;
    res = color;     /* Non-compliant */
}
```

In this example, the rule is violated because there is an unreachable operation following the `return` statement.

### Dead code

#### Issue

The issue occurs when the analysis detects a reachable operation that does not affect program behavior if the operation is removed.

Polyspace Bug Finder detects useless write operations during analysis.

#### Risk

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

Operations involving language extensions such as `__asm ( "NOP" );` are not considered dead code.

#### Example - Redundant Operations

```
extern volatile unsigned int v;
extern char *p;

void f ( void ) {
    unsigned int x;


    ( void ) v;      /* Compliant - Exception*/
    ( int ) v;       /* Non-compliant  */
    v >> 3;          /* Non-compliant  */
    x = 3;           /* Non-compliant - Detected in Bug Finder only */

    *p++;            /* Non-compliant  */
    ( *p )++;        /* Compliant  */
}
```

In this example, the rule is violated when an operation is performed on a variable, but the result of that operation is not used. For instance,

- The operations (int) and >> on the variable v are redundant because the results are not used.
- The operation = is redundant because the local variable x is not read after the operation.
- The operation * on p++ is redundant because the result is not used.

The rule is not violated when:

- A variable is cast to void. The cast indicates that you are intentionally not using the value.
- The result of an operation is used. For instance, the operation * on p is not redundant, because *p is incremented.

### Useless if

#### Issue

This issue occurs on if-statements where the condition is always true. This defect occurs only on if-statements that do not have an else-statement.

This defect shows unnecessary if-statements when there is no difference in code execution if the if-statement is removed.

#### Risk

Unnecessary if statements often indicate a coding error. Perhaps the if condition is coded incorrectly or the if statement is not required at all.

#### Fix

The fix depends on the root cause of the defect. For instance, the root cause can be an error condition that is checked twice on the same execution path, making the second check redundant.

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If the redundant condition represents defensive coding practices and you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.

- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Example – if with Enumerated Type

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card < 7) { //Noncompliant
        do_something(card);
    }
}
```

The type `suit` is enumerated with five options. However, the conditional expression `card < 7` always evaluates to true because `card` can be at most 5. The `if` statement is unnecessary.

### Correction 1 — Change Condition

One possible correction is to change the if-condition in the code. In this correction, the 7 is changed to `UNKNOWN_SUIT` to relate directly to the type of `card`.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card > UNKNOWN_SUIT) {
        do_something(card);
    }
}
```

### Correction — Remove If

Another possible correction is to remove the if-condition in the code. Because the condition is always true, you can remove the condition to simplify your code.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }
    do_something(card);
}
```

### Write without a further read

#### Issue

This issue occurs when a value assigned to a variable is never read.

For instance, you write a value to a variable and then write a second value before reading the previous value. The first write operation is redundant.

#### Risk

Redundant write operations often indicate programming errors. For instance, you forgot to read the variable between two successive write operations or unintentionally read a different variable.

#### Fix

Identify the reason why you write to the variable but do not read it later. Look for common programming errors such as accidentally reading a different variable with a similar name.

If you determine that the write operation is redundant, remove the operation.

#### Example – Write Without Further Read Error

```
void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();             //Noncompliant
    /* Defect: Useless write */
}
```

After the variable `level` gets assigned the value `4 * getsensor()`, it is not read.

#### Correction — Use Value After Assignment

One possible correction is to use the variable `level` after the assignment.

```
#include <stdio.h>

void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();

    /* Fix: Use level after assignment */
    printf("The value is %d", level);

}
```

The variable `level` is printed, reading the new value.

## Check Information

**Group:** Rec. 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MSC12-C

# CERT C: Rec. MSC13-C

Detect and remove unused values

## Description

### Rule Definition

*Detect and remove unused values.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Unused parameter**.
- **Write without a further read**.

## Examples

### Unused parameter

#### Issue

**Unused parameter** occurs when a function parameter is neither read nor written in the function body.

#### Risk

Unused function parameters cause the following issues:

- Indicate that the code is possibly incomplete. The parameter is possibly intended for an operation that you forgot to code.
- If the copied objects are large, redundant copies can slow down performance.

#### Fix

Determine if you intend to use the parameters. Otherwise, remove parameters that you do not use in the function body.

You can intentionally have unused parameters. For instance, you have parameters that you intend to use later when you add enhancements to the function. Add a code comment indicating your intention for later use. The code comment helps you or a code reviewer understand why your function has unused parameters.

Alternatively, add a statement such as `(void)var;` in the function body. `var` is the unused parameter. You can define a macro that expands to this statement and add the macro to the function body.

#### Example - Unused Parameter

```
void func(int* xptr, int* yptr, int flag) { //Noncompliant
    if(flag==1) {
        *xptr=0;
    }
    else {
        *xptr=1;
    }
}

int main() {
    int x,y;
    func(&x,&y,1);
    return 0;
}
```

In this example, the parameter `yptr` is not used in the body of `func`.

#### Correction — Use Parameter

One possible correction is to check if you intended to use the parameter. Fix your code if you intended to use the parameter.

```
void func(int* xptr, int* yptr, int flag) {
    if(flag==1) {
        *xptr=0;
        *yptr=1;
    }
    else {
        *xptr=1;
        *yptr=0;
    }
}

int main() {
    int x,y;
    func(&x,&y,1);
    return 0;
}
```

#### Correction — Explicitly Indicate Unused Parameter

Another possible correction is to explicitly indicate that you are aware of the unused parameter.

```
#define UNUSED(x) (void)x

void func(int* xptr, int* yptr, int flag) {
    UNUSED(yptr);
    if(flag==1) {
        *xptr=0;
    }
    else {
        *xptr=1;
    }
}

int main() {
    int x,y;
    func(&x,&y,1);
    return 0;
}
```

### Write without a further read

#### Issue

**Write without a further read** occurs when a value assigned to a variable is never read.

For instance, you write a value to a variable and then write a second value before reading the previous value. The first write operation is redundant.

#### Risk

Redundant write operations often indicate programming errors. For instance, you forgot to read the variable between two successive write operations or unintentionally read a different variable.

#### Fix

Identify the reason why you write to the variable but do not read it later. Look for common programming errors such as accidentally reading a different variable with a similar name.

If you determine that the write operation is redundant, remove the operation.

#### Example - Write Without Further Read Error

```
void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();             //Noncompliant
    /* Defect: Useless write */
}
```

After the variable `level` gets assigned the value `4 * getsensor()`, it is not read.

#### Correction — Use Value After Assignment

One possible correction is to use the variable `level` after the assignment.

```
#include <stdio.h>

void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();

    /* Fix: Use level after assignment */
    printf("The value is %d", level);

}
```

The variable `level` is printed, reading the new value.

## Check Information

**Group:** Rec. 48. Miscellaneous (MSC)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC13-C

# CERT C: Rec. MSC15-C

Do not depend on undefined behavior

## Description

### Rule Definition

*Do not depend on undefined behavior.*

### Polyspace Implementation

The rule checker checks for **Undefined behavior**.

## Examples

### Undefined behavior

#### Issue

The issue occurs when the analysis detects undefined or critical unspecified behaviour. Specifically, Polyspace flags these instances of undefined or critical undefined behavior:

- Use of `offsetof` on bit fields.
- Use of `offsetof` when the second argument is not a `struct` field of the first argument.
- Use of `defined` without an identifier.
- Use of an array of incomplete types.
- Use of a function like macros by using incorrect number of arguments.

#### Risk

C code that results in undefined or critical unspecified behavior might produce unexpected or incorrect results. Such code might behave differently in different implementations. Issues caused by undefined behavior in the code might be difficult to analyze because compilers might optimize the code assuming that undefined behavior does not occur.

#### Fix

Avoid code that might result in undefined or critically unspecified behavior.

#### Example — Avoid Undefined Behaviors

```
#include <stddef.h>     /* offsetof */

struct str {
  char a:8;
  char b[10];
  char c;
};
void foo() {

  offsetof(struct str, a);//Noncompliant
  offsetof(struct str, d);//Noncompliant
}
```

In this example, the function `foo` uses the macro `offsetof` on the bit field `str.a`. This behavior is undefined. Polyspace flags it. The function then calls `offsetof` on `str.d`. Because `d` is not a field of `str`, Polyspace flags it.

These issues might cause compilation errors with your compiler. Polyspace flags this issue as a rule violation while also showing a compilation failure.

## Check Information

**Group:** Rec. 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC15-C

# CERT C: Rec. MSC17-C

Finish every set of statements associated with a case label with a break statement

## Description

### Rule Definition

*Finish every set of statements associated with a case label with a break statement.*

### Polyspace Implementation

The rule checker checks for **Missing break of switch case**.

## Examples

### Missing break of switch case

#### Issue

**Missing break of switch case** looks for switch cases that do not end in a `break` statement.

If the last entry in the case block is a code comment, for instance:

```
switch (wt)
    {
      case WE_W:
        do_something_for_WE_W();
        do_something_else_for_WE_W();
        /* fall through to WE_X*/
      case WE_X:
        ...
    }
```

Polyspace assumes that the missing break is intentional and does not raise a defect.

#### Risk

Switch cases without break statements fall through to the next switch case. If this fall-through is not intended, the switch case can unintentionally execute code and end the switch with unexpected results.

#### Fix

If you do not want a break for the highlighted switch case, add a comment to your code to document why this case falls through to the next case. This comment removes the defect from your results and makes your code more maintainable.

If you forgot the break, add it before the end of the switch case.

#### Example - Switch Without Break Statements

```
enum WidgetEnum { WE_W, WE_X, WE_Y, WE_Z } widget_type;

extern void demo_do_something_for_WE_W(void);
extern void demo_do_something_for_WE_X(void);
extern void demo_report_error(void);

void bug_missingswitchbreak(enum WidgetEnum wt)
{
    /*
      In this non-compliant code example, the case where widget_type is WE_W lacks a
      break statement. Consequently, statements that should be executed only when
      widget_type is WE_X are executed even when widget_type is WE_W.
    */
    switch (wt)
    {
      case WE_W:  //Noncompliant
        demo_do_something_for_WE_W();
      case WE_X:  //Noncompliant
        demo_do_something_for_WE_X();
      default:
        /* Handle error condition */
        demo_report_error();
    }
}
```

In this example, there are two cases without `break` statements. When `wt` is `WE_W`, the statements for `WE_W`, `WE_X`, and the `default` case execute because the program falls through the two cases without a break. No defect is raised on the `default` case or last case because it does not need a break statement.

**Correction — Add a Comment or break**

To fix this example, either add a comment to mark and document the acceptable fall-through or add a break statement to avoid fall-through. In this example, case `WE_W` is supposed to fall through, so a comment is added to explicitly state this action. For the second case, a break statement is added to avoid falling through to the `default` case.

```
enum WidgetEnum { WE_W, WE_X, WE_Y, WE_Z } widget_type;

extern void demo_do_something_for_WE_W(void);
extern void demo_do_something_for_WE_X(void);
extern void demo_report_error(void);

void corrected_missingswitchbreak(enum WidgetEnum wt)
{
    switch (wt)
    {
      case WE_W:
        demo_do_something_for_WE_W();
        /* fall through to WE_X*/
      case WE_X:
        demo_do_something_for_WE_X();
        break;
      default:
        /* Handle error condition */
        demo_report_error();
    }
}
```

## Check Information

**Group:** Rec. 48. Miscellaneous (MSC)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC17-C

# CERT C: Rec. MSC18-C

Be careful while handling sensitive data, such as passwords, in program code

## Description

### Rule Definition

*Be careful while handling sensitive data, such as passwords, in program code.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Constant block cipher initialization vector**.
- **Constant cipher key**.
- **Predictable block cipher initialization vector**.
- **Predictable cipher key**.
- **Sensitive heap memory not cleared before release**.
- **Uncleared sensitive data in stack**.
- **Unsafe standard encryption function**.

## Examples

### Constant block cipher initialization vector

#### Issue

**Constant block cipher initialization vector** occurs when you use a constant for the initialization vector (IV) during encryption.

#### Risk

Using a constant IV is equivalent to not using an IV. Your encrypted data is vulnerable to dictionary attacks.

Block ciphers break your data into blocks of fixed size. Block cipher modes such as CBC (Cipher Block Chaining) protect against dictionary attacks by XOR-ing each block with the encrypted output from the previous block. To protect the first block, these modes use a random initialization vector (IV). If you use a constant IV to encrypt multiple data streams that have a common beginning, your data becomes vulnerable to dictionary attacks.

#### Fix

Produce a random IV by using a strong random number generator.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

#### Example - Constants Used for Initialization Vector

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

/* Using the cryptographic routines */

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16] = {'1', '2', '3', '4','5','6','b','8','9',
                                 '1','2','3','4','5','6','7'};
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);  //Noncompliant
}
```

In this example, the initialization vector `iv` has constants only. The constant initialization vector makes your cipher vulnerable to dictionary attacks.

#### Correction — Use Random Initialization Vector

One possible correction is to use a strong random number generator to produce the initialization vector. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

/* Using the cryptographic routines */

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

### Constant cipher key

#### Issue

**Constant cipher key** occurs when you use a constant for the encryption or decryption key.

#### Risk

If you use a constant for the encryption or decryption key, an attacker can retrieve your key easily.

You use a key to encrypt and later decrypt your data. If a key is easily retrieved, data encrypted using that key is not secure.

#### Fix

Produce a random key by using a strong random number generator.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

#### Example - Constants Used for Key

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16] = {'1', '2', '3', '4','5','6','b','8','9',
                                 '1','2','3','4','5','6','7'};
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);  //Noncompliant
}
```

In this example, the cipher key, `key`, has constants only. An attacker can easily retrieve a constant key.

#### Correction — Use Random Key

Use a strong random number generator to produce the cipher key. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

### Predictable block cipher initialization vector

#### Issue

**Predictable block cipher initialization vector** occurs when you use a weak random number generator for the block cipher initialization vector.

#### Risk

If you use a weak random number generator for the initiation vector, your data is vulnerable to dictionary attacks.

Block ciphers break your data into blocks of fixed size. Block cipher modes such as CBC (Cipher Block Chaining) protect against dictionary attacks by XOR-ing each block with the encrypted output from the previous block. To protect the first block, these modes use a random initialization vector (IV). If you use a weak random number generator for your IV, your data becomes vulnerable to dictionary attacks.

#### Fix

Use a strong pseudo-random number generator (PRNG) for the initialization vector. For instance, use:

- OS-level PRNG such as `/dev/random` on UNIX or `CryptGenRandom()` on Windows
- Application-level PRNG such as Advanced Encryption Standard (AES) in Counter (CTR) mode, HMAC-SHA1, etc.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

#### Example - Predictable Initialization Vector

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_pseudo_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);  //Noncompliant
}
```

In this example, the function `RAND_pseudo_bytes` declared in `openssl/rand.h` produces the initialization vector. The byte sequences that `RAND_pseudo_bytes` generates are not necessarily unpredictable.

#### Correction — Use Strong Random Number Generator

Use a strong random number generator to produce the initialization vector. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

### Predictable cipher key

#### Issue

**Predictable cipher key** occurs when you use a weak random number generator for the encryption or decryption key.

#### Risk

If you use a weak random number generator for the encryption or decryption key, an attacker can retrieve your key easily.

You use a key to encrypt and later decrypt your data. If a key is easily retrieved, data encrypted using that key is not secure.

#### Fix

Use a strong pseudo-random number generator (PRNG) for the key. For instance:

- Use an OS-level PRNG such as `/dev/random` on UNIX or `CryptGenRandom()` on Windows
- Use an application-level PRNG such as Advanced Encryption Standard (AES) in Counter (CTR) mode, HMAC-SHA1, etc.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

#### Example - Predictable Cipher Key

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_pseudo_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);  //Noncompliant
}
```

In this example, the function `RAND_pseudo_bytes` declared in `openssl/rand.h` produces the cipher key. However, the byte sequences that `RAND_pseudo_bytes` generates are not necessarily unpredictable.

#### Correction — Use Strong Random Number Generator

One possible correction is to use a strong random number generator to produce the cipher key. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

### Sensitive heap memory not cleared before release

#### Issue

**Sensitive heap memory not cleared before release** detects dynamically allocated memory containing sensitive data. If you do not clear the sensitive data when you free the memory, Bug Finder raises a defect on the `free` function.

#### Risk

If the memory zone is reallocated, an attacker can still inspect the sensitive data in the old memory zone.

#### Fix

Before calling `free`, clear out the sensitive data using `memset` or `SecureZeroMemory`.

#### Example - Sensitive Buffer Freed, Not Cleared

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    free(buf); //Noncompliant
}
```

In this example, the function uses a buffer of passwords and frees the memory before the end of the function. However, the data in the memory is not cleared by using the `free` command.

#### Correction — Nullify Data

One possible correction is to write over the data to clear out the sensitive information. This example uses `memset` to write over the data with zeros.

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0;i<(sizeof(arr)/sizeof(arr[0]));i++) assert(arr[i]==0)

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);

    if (buf) {
        getpwnam_r(my_user, &pwd, buf, bufsize, &result);
        memset(buf, 0, (size_t)1024);
        isNull(buf);
        free(buf);
    }
}
```

### Uncleared sensitive data in stack

#### Issue

**Uncleared sensitive data in stack** detects static memory containing sensitive data. If you do not clear the sensitive data from your stack before exiting the function or program, Bug Finder raises a defect on the last curly brace.

#### Risk

Leaving sensitive information in your stack, such as passwords or user information, allows an attacker additional access to the information after your program has ended.

#### Fix

Before exiting a function or program, clear out the memory zones that contain sensitive data by using `memset` or `SecureZeroMemory`.

#### Example - Static Buffer of Password Information

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>

void bug_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
}  //Noncompliant
```

In this example, a static buffer is filled with password information. The program frees the stack memory at the end of the program. However, the data is still accessible from the memory.

#### Correction — Clear Memory

One possible correction is to write over the memory before exiting the function. This example uses `memset` to clear the data from the buffer memory.

```
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0; i<(sizeof(arr)/sizeof(arr[0])); i++) assert(arr[i]==0)

void corrected_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    memset(buf, 0, (size_t)1024);
    isNull(buf);
}
```

### Unsafe standard encryption function

#### Issue

**Unsafe standard encryption function** detects use of functions with a broken or weak cryptographic algorithm. For example, `crypt` is not reentrant and is based on the risky Data Encryption Standard (DES).

#### Risk

The use of a broken, weak, or nonstandard algorithm can expose sensitive information to an attacker. A determined hacker can access the protected data using various techniques.

If the weak function is nonreentrant, when you use the function in concurrent programs, there is an additional race condition risk.

#### Fix

Avoid functions that use these encryption algorithms. Instead, use a reentrant function that uses a stronger encryption algorithm.

**Note** Some implementations of `crypt` support additional, possibly more secure, encryption algorithms.

#### Example - Decrypting Password Using `crypt`

```
#define _GNU_SOURCE
#include <pwd.h>
#include <string.h>
#include <crypt.h>

volatile int rd = 1;

const char *salt = NULL;
struct crypt_data input, output;

int verif_pwd(const char *pwd, const char *cipher_pwd, int safe)
{
    int r = 0;
    char *decrypted_pwd = NULL;

    switch(safe)
    {
      case 1:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      case 2:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      default:
        decrypted_pwd = crypt(pwd, cipher_pwd);  //Noncompliant
        break;
    }

    r = (strcmp(cipher_pwd, decrypted_pwd) == 0);

    return r;
}
```

In this example, `crypt_r` and `crypt` decrypt a password. However, `crypt` is nonreentrant and uses the unsafe Data Encryption Standard algorithm.

#### Correction — Use `crypt_r`

One possible correction is to replace `crypt` with `crypt_r`.

```
#define _GNU_SOURCE
#include <pwd.h>
```

```
#include <string.h>
#include <crypt.h>

volatile int rd = 1;

const char *salt = NULL;
struct crypt_data input, output;

int verif_pwd(const char *pwd, const char *cipher_pwd, int safe)
{
    int r = 0;
    char *decrypted_pwd = NULL;

    switch(safe)
    {
      case 1:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      case 2:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      default:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;
    }

    r = (strcmp(cipher_pwd, decrypted_pwd) == 0);

    return r;
}
```

## Check Information

**Group:** Rec. 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C (-cert-c)

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MSC18-C

# CERT C: Rec. MSC20-C

Do not use a switch statement to transfer control into a complex block

## Description

### Rule Definition

*Do not use a switch statement to transfer control into a complex block.*

### Polyspace Implementation

The rule checker checks for **Switch label not at outermost level of body of switch statement**.

## Examples

### Switch label not at outermost level of body of switch statement

#### Issue

The issue occurs when you use a switch label and the most closely-enclosing compound statement is not the body of the switch statement. For instance a `case` label is enclosed inside a `for` loop that is enclosed inside the switch statement.

#### Risk

The C Standard permits placing a switch label (for instance, `case` or `default`) before any statement contained in the body of a switch statement. This flexibility can lead to unstructured code. To prevent unstructured code, make sure a switch label appears only at the outermost level of the body of a switch statement.

## Check Information

**Group:** Rec. 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC20-C

# CERT C: Rec. MSC21-C

Use robust loop termination conditions

## Description

### Rule Definition

*Use robust loop termination conditions.*

### Polyspace Implementation

The rule checker checks for **Loop bounded with tainted value**.

### Extend Checker

A default Bug Finder analysis might not flag a **Loop bounded with tainted value** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Loop bounded with tainted value

#### Issue

**Loop bounded with tainted value** detects loops that are bounded by values from an unsecure source.

#### Risk

A tainted value can cause over looping or infinite loops. Attackers can use this vulnerability to crash your program or cause other unintended behavior.

#### Fix

Before starting the loop, validate unknown boundary and iterator values.

#### Example — Loop Boundary From User Input

```
#include<stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedloopboundary(void) {
    int count;
    scanf("%d", &count);
    int res = 0;
    for (int i=0 ; i < count; ++i) { //Noncompliant
        res += i;
    }
    return res;
}
```

In this example, the function uses a user input to loop `count` times. `count` could be any number because the value is not checked before starting the `for` loop.

#### Correction: Clamp Tainted Loop Control

One possible correction is to clamp the tainted loop control. To validate the tainted loop variable `count`, this example limits `count` to a minimum value and a maximum value by using inline functions `min` and `max`. Regardless of the user input, the value of `count` remains within a known range.

```
#include<stdio.h>
#include<algorithm>
#define MIN 50
#define MAX 128
static  inline int max(int a, int b) { return a > b ? a : b;}
static inline int min(int a, int b) { return a < b ? a : b; }

int taintedloopboundary(void) {
    int count;
    scanf("%d", &count);
    int res = 0;
    count = max(MIN, min(count, MAX));
    for (int i=0 ; i<count ; ++i) {
        res += i;
    }
    return res;
}
```

#### Correction — Check Tainted Loop Control

Another possible correction is to check the low bound and the high bound of the tainted loop boundary variable before starting the `for` loop. This example checks the low and high bounds of `count` and executes the loop only when `count` is between 0 and 127.

```
#include<stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};


int taintedloopboundary(void) {
    int count;
    scanf("%d", &count);
    int res = 0;

    if (count>=0 && count<SIZE128) {
        for (int i=0 ; i<count ; ++i) {
            res += i;
        }
    }
    return res;
}
```

## Check Information

**Group:** Rec. 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC21-C

# CERT C: Rec. MSC22-C

Use the setjmp(), longjmp() facility securely

## Description

### Rule Definition

*Use the setjmp(), longjmp() facility securely.*

### Polyspace Implementation

The rule checker checks for **Use of setjmp/longjmp**.

## Examples

### Use of setjmp/longjmp

#### Issue

**Use of setjmp/longjmp** occurs when you use a combination of `setjmp` and `longjmp` or `sigsetjmp` and `siglongjmp` to deviate from normal control flow and perform non-local jumps in your code.

#### Risk

Using `setjmp` and `longjmp`, or `sigsetjmp` and `siglongjmp` has the following risks:

- Nonlocal jumps are vulnerable to attacks that exploit common errors such as buffer overflows. Attackers can redirect the control flow and potentially execute arbitrary code.
- Resources such as dynamically allocated memory and open files might not be closed, causing resource leaks.
- If you use `setjmp` and `longjmp` in combination with a signal handler, unexpected control flow can occur. POSIX does not specify whether `setjmp` saves the signal mask.
- Using `setjmp` and `longjmp` or `sigsetjmp` and `siglongjmp` makes your program difficult to understand and maintain.

#### Fix

Perform nonlocal jumps in your code using `setjmp/longjmp` or `sigsetjmp/siglongjmp` only in contexts where such jumps can be performed securely. Alternatively, use POSIX threads if possible.

In C++, to simulate throwing and catching exceptions, use standard idioms such as `throw` expressions and `catch` statements.

#### Example - Use of `setjmp` and `longjmp`

```
#include <setjmp.h>
#include <signal.h>

extern int update(int);
extern void print_int(int);

static jmp_buf env;
```

```
void sighandler(int signum) {
    longjmp(env, signum); //Noncompliant
}
void func_main(int i) {
    signal(SIGINT, sighandler);
    if (setjmp(env)==0) { //Noncompliant
        while(1) {
            /* Main loop of program, iterates until SIGINT signal catch */
            i = update(i);
        }
    } else {
        /* Managing longjmp return */
        i = -update(i);
    }

    print_int(i);
    return;
}
```

In this example, the initial return value of `setjmp` is 0. The `update` function is called in an infinite `while` loop until the user interrupts it through a signal.

In the signal handling function, the `longjmp` statement causes a jump back to `main` and the return value of `setjmp` is now 1. Therefore, the `else` branch is executed.

**Correction — Use Alternative to `setjmp` and `longjmp`**

To emulate the same behavior more securely, use a `volatile` global variable instead of a combination of `setjmp` and `longjmp`.

```
#include <setjmp.h>
#include <signal.h>

extern int update(int);
extern void print_int(int);

volatile sig_atomic_t eflag = 0;

void sighandler(int signum) {
     eflag = signum;                   /* Fix: using global variable */
}

void func_main(int i) {
      /* Fix: Better design to avoid use of setjmp/longjmp */
    signal(SIGINT, sighandler);
    while(!eflag) {                   /* Fix: using global variable */
        /* Main loop of program, iterates until eflag is changed */
        i = update(i);
    }

    print_int(i);
    return;
}
```

## Check Information

**Group:** Rec. 48. Miscellaneous (MSC)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC22-C

# CERT C: Rec. MSC24-C

Do not use deprecated or obsolescent functions

## Description

### Rule Definition

*Do not use deprecated or obsolescent functions.*

### Polyspace Implementation

The rule checker checks for **Use of obsolete standard function**.

## Examples

### Use of obsolete standard function

#### Issue

**Use of obsolete standard function** detects calls to standard function routines that are considered legacy, removed, deprecated, or obsolete by C/C++ coding standards.

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| `asctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |
| `asctime_r` | Deprecated in POSIX.1-2008 | Implementation based on unsafe function `sprintf`. | `strftime` or `asctime_s` |
| `bcmp` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcmp` |
| `bcopy` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcpy` or `memmove` |
| `brk` and `sbrk` | Marked as legacy in SUSv2 and POSIX.1-2001. | | `malloc` |
| `bsd_signal` | Removed in POSIX.1-2008 | | `sigaction` |
| `bzero` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | `memset` |
| `ctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| ctime_r | Deprecated in POSIX.1-2008 | Implementation based on unsafe function sprintf. | strftime or asctime_s |
| cuserid | Removed in POSIX.1-2001. | Not reentrant. Precise functionality not standardized causing portability issues. | getpwuid |
| ecvt and fcvt | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | Not reentrant | snprintf |
| ecvt_r and fcvt_r | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | | snprintf |
| ftime | Removed in POSIX.1-2008 | | time, gettimeofday, clock_gettime |
| gamma, gammaf, gammal | Function not specified in any standard because of historical variations | Portability issues. | tgamma, lgamma |
| gcvt | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | snprintf |
| getcontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| getdtablesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_OPEN _MAX ) |
| gethostbyaddr | Removed in POSIX.1-2008 | Not reentrant | getaddrinfo |
| gethostbyname | Removed in POSIX.1-2008 | Not reentrant | getnameinfo |
| getpagesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_PAGE SIZE ) |
| getpass | Removed in POSIX.1-2001. | Not reentrant. | getpwuid |
| getw | Not present in POSIX.1-2001. | | fread |
| getwd | Marked legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | getcwd |
| index | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strchr |
| makecontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| memalign | Appears in SunOS 4.1.3. Not in 4.4 BSD or POSIX.1-2001 | | posix_memalign |
| mktemp | Removed in POSIX.1-2008. | Generated names are predictable and can cause a race condition. | mkstemp removes race risk |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| pthread_attr_ getstackaddr and pthread_attr_ setstackaddr | | Ambiguities in the specification of the stackaddr attribute cause portability issues | pthread_attr_ getstack and pthread_attr_ setstack |
| putw | Not present in POSIX.1-2001. | Portability issues. | fwrite |
| qecvt and qfcvt | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | snprintf |
| qecvt_r and qfcvt_r | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | snprintf |
| rand_r | Marked as obsolete in POSIX.1-2008 | | |
| re_comp | BSD API function | Portability issues | regcomp |
| re_exes | BSD API function | Portability issues | regexec |
| rindex | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strrchr |
| scalb | Removed in POSIX.1-2008 | | scalbln, scalblnf, or scalblnl |
| sigblock | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigmask | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigsetmask | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigstack | Interface is obsolete and not implemented on most platforms. | Portability issues. | sigaltstack |
| sigvec | 4.3BSD signal API whose origin is unclear | | sigaction |
| swapcontext | Removed in POSIX.1-2008 | Portability issues. | Use POSIX threads. |
| tmpnam and tmpnam_r | Marked as obsolete in POSIX.1-2008. | This function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined. | mkstemp, tmpfile |
| ttyslot | Removed in POSIX.1-2001. | | |
| ualarm | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | Errors are under-specified | setitimer or POSIX timer_create |
| usleep | Removed in POSIX.1-2008. | | nanosleep |
| utime | SVr4, POSIX.1-2001. POSIX.1-2008 marks as obsolete. | | |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| valloc | Marked as obsolete in 4.3BSD.<br>Marked as legacy in SUSv2.<br>Removed from POSIX.1-2001 | | posix_memalign |
| vfork | Removed from POSIX.1-2008 | Under-specified in previous standards. | fork |
| wcswcs | This function was not included in the final ISO/IEC 9899:1990/ Amendment 1:1995 (E). | | wcsstr |
| WinExec | WinAPI provides this function only for 16-bit Windows compatibility. | | CreateProcess |
| LoadModule | WinAPI provides this function only for 16-bit Windows compatibility. | | CreateProcess |

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Printing Out Time

```
#include <stdio.h>
#include <time.h>

void timecheck_bad(int argc, char *argv[])
{
    time_t ticks;

    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks)); //Noncompliant
}
```

In this example, the function ctime formats the current time and prints it out. However, ctime was removed after C99 because it does not work on multithreaded programs.

#### Correction — Different Time Function

One possible correction is to use strftime instead because this function uses a set buffer size.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

void timecheck_good(int argc, char *argv[])
{
    char outBuff[1025];
    time_t ticks;
    struct tm * timeinfo;

    memset(outBuff, 0, sizeof(outBuff));

    ticks = time(NULL);
    timeinfo = localtime(&ticks);
    strftime(outBuff,sizeof(outBuff),"%I:%M%p.",timeinfo);
    fprintf(stdout, outBuff);
}
```

## Check Information

**Group:** Rec. 48. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC24-C

# CERT C: Rec. POS05-C

Limit access to files by creating a jail

## Description

### Rule Definition

*Limit access to files by creating a jail.*

### Polyspace Implementation

The rule checker checks for **File manipulation after chroot without chdir**.

## Examples

### File manipulation after chroot without chdir

#### Issue

**File manipulation after `chroot()` without `chdir("/")`** detects access to the file system outside of the jail created by `chroot`. By calling `chroot`, you create a file system jail that confines access to a specific file subsystem. However, this jail is ineffective if you do not call `chdir("/")`.

#### Risk

If you do not call `chdir("/")` after creating a `chroot` jail, file manipulation functions that takes a path as an argument can access files outside of the jail. An attacker can still manipulate files outside the subsystem that you specified, making the chroot jail ineffective.

#### Fix

After calling `chroot`, call `chdir("/")` to make your `chroot` jail more secure.

#### Example - Open File in `chroot`-jail

```
#include <unistd.h>
#include <stdio.h>

const char root_path[] = "/var/ftproot";
const char log_path[] = "file.log";
FILE* chrootmisuse() {
    FILE* res;
    chroot(root_path);
    chdir("base");  //Noncompliant
    res = fopen(log_path, "r");  //Noncompliant
    return res;
}
```

This example uses `chroot` to create a chroot-jail. However, to use the `chroot` jail securely, you must call `chdir("\")` afterward. This example calls `chdir("base")`, which is not equivalent. Bug Finder also flags `fopen` because `fopen` opens a file in the vulnerable `chroot`-jail.

#### Correction — Call `chdir("/")`

Before opening files, call `chdir("/")`.

```
#include <unistd.h>
#include <stdio.h>

const char root_path[] = "/var/ftproot";
const char log_path[] = "file.log";
FILE* chrootmisuse() {
    FILE* res;
    chroot(root_path);
    chdir("/");
    res = fopen(log_path, "r");
    return res;
}
```

## Check Information

**Group:** Rec. 50. POSIX (POS)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

POS05-C

# CERT C: Rec. WIN00-C

Be specific when dynamically loading libraries

## Description

### Rule Definition

*Be specific when dynamically loading libraries.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Load of library from a relative path can be controlled by an external actor**.
- **Library loaded from externally controlled path**.

### Extend Checker

A default Bug Finder analysis might not flag a **Library loaded from externally controlled path** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Load of library from a relative path can be controlled by an external actor

#### Issue

**Load of library from a relative path can be controlled by an external actor** detects library loading routines that load an external library. If you load the library using a relative path or no path, Bug Finder flags the loading routine as a defect.

#### Risk

By using a relative path or no path to load an external library, your program uses an unsafe search process to find the library. An attacker can control the search process and replace the intended library with a library of their own.

#### Fix

When you load an external library, specify the full path.

#### Example - Open Library with Library Name

```
#include <dlfcn.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>

void relative_path()
```

```
{
    dlopen("liberty.dll",RTLD_LAZY); //Noncompliant
}
```

In this example, `dlopen` opens the `liberty` library by calling only the name of the library. However, this call to the library uses a relative path to find the library, which is unsafe.

#### Correction — Use Full Path to Library

One possible correction is to use the full path to the library when you load it into your program.

```
#include <dlfcn.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>

void relative_path()
{
    dlopen("/home/my_libs/library/liberty.dll",RTLD_LAZY);
}
```

### Library loaded from externally controlled path

#### Issue

**Library loaded from externally controlled path** looks for libraries loaded from fixed or controlled paths. If unintended actors can control one or more locations on this fixed path, Bug Finder raises a defect.

#### Risk

If an attacker knows or controls the path that you use to load a library, the attacker can change:

- The library that the program loads, replacing the intended library and commands.
- The environment in which the library executes, giving unintended permissions and capabilities to the attacker.

#### Fix

When possible, use hard-coded or fully qualified path names to load libraries. It is possible the hard-coded paths do not work on other systems. Use a centralized location for hard-coded paths, so that you can easily modify the path within the source code.

Another solution is to use functions that require explicit paths. For example, `system()` does not require a full path because it can use the `PATH` environment variable. However, `execl()` and `execv()` do require the full path.

#### Example - Call Custom Library

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void* taintedpathlib() {
    void* libhandle = NULL;
    char lib[SIZE128] = "";
    char* userpath = getenv("LD_LIBRARY_PATH");
    strncpy(lib, userpath, SIZE128);
    strcat(lib, "/libX.so");
    libhandle = dlopen(lib, 0x00001); //Noncompliant
    return libhandle;
}
```

This example loads the library `libX.so` from an environment variable `LD_LIBRARY_PATH`. An attacker can change the library path in this environment variable. The actual library you load could be a different library from the one that you intend.

**Correction — Change and Check Path**

One possible correction is to change how you get the library path and check the path of the library before opening the library. This example receives the path as an input argument. Then the path is checked to make sure the library is not under `/usr/`.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

/* Function to sanitize a string */
int sanitize_str(char* s, size_t n) {
    /* strlen is used here as a kind of firewall for tainted string errors */
    int res = (strlen(s) > 0 && strlen(s) < n);
    return res;
}
void* taintedpathlib(char* userpath) {
    void* libhandle = NULL;
    if (sanitize_str(userpath, SIZE128)) {
        char lib[SIZE128] = "";

        if (strncmp(userpath, "/usr", 4)!=0) {
            strncpy(lib, userpath, SIZE128);
            strcat(lib, "/libX.so");
            libhandle = dlopen(lib, RTLD_LAZY);
        }
    }
    return libhandle;
}
```

## Check Information

**Group:** Rec. 51. Microsoft Windows (WIN)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C (-cert-c)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

WIN00-C

23

# ISO/IEC TS 17961

# Acknowledgment

Extracts from the standard "ISO/IEC TS 17961 Technical Specification - 2013-11-15" are reproduced with the agreement of AFNOR. Only the original and complete text of the standard, as published by AFNOR Editions - accessible via the website www.boutique.afnor.org - has normative value.

# ISO/IEC TS 17961 [accfree]

Accessing freed memory

## Description

### Rule Definition

*Accessing freed memory.*

### Polyspace Implementation

This checker checks for these issues:

- **Use of previously freed pointer**.
- **Invalid use of standard library string routine**.

## Examples

### Use of previously freed pointer

#### Issue

**Use of previously freed pointer** occurs when you access a block of memory after freeing the block using the `free` function.

#### Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

#### Example - Use of Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   { 
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);
    j = *pi + shift;
    /* Defect: Reading a freed pointer */

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

**Correction — Free Pointer After Use**

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

### Invalid use of standard library string routine

**Issue**

**Invalid use of standard library string routine** occurs when a string library function is called with invalid arguments.

**Risk**

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

**Fix**

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source);
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Invalid Use of Standard Library String Routine Error**

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

**Correction — Use Valid Arguments**

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

**Topics**
“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [accsig]

Accessing shared objects in signal handlers

## Description

### Rule Definition

*Accessing shared objects in signal handlers.*

### Polyspace Implementation

This checker checks for **Shared data access within signal handler**.

## Examples

### Shared data access within signal handler

#### Issue

**Shared data access within signal handler** occurs when you access or modify a shared object inside a signal handler.

#### Risk

When you define a signal handler function to access or modify a shared object, the handler accesses or modifies the shared object when it receives a signal. If another function is already accessing the shared object, that function causes a race condition and can leave the data in an inconsistent state.

#### Fix

To access or modify shared objects inside a signal handler, check that the objects are lock-free atomic, or, if they are integers, declare them as `volatile sig_atomic_t`.

#### Example - `int` Variable Access in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* declare global variable. */
int e_flag;

void sig_handler(int signum)
{
    /* Signal handler accesses variable that is not
    of type volatile sig_atomic_t. */
    e_flag = signum;
}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` accesses `e_flag`, a variable of type `int`. A concurrent access by another function can leave `e_flag` in an inconsistent state.

### Correction — Declare Variable of Type `volatile sig_atomic_t`

Before you access a shared variable from a signal handler, declare the variable with type `volatile sig_atomic_t` instead of `int`. You can safely access variables of this type asynchronously.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* Declare variable of type volatile sig_atomic_t. */
volatile sig_atomic_t e_flag;
void sig_handler(int signum)
{
    /* Use variable of proper type inside signal handler. */
    e_flag = signum;

}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

## Check Information

**Decidability:** Undecidable

## Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [addrescape]

Escaping of the address of an automatic object

## Description

### Rule Definition

*Escaping of the address of an automatic object.*

### Polyspace Implementation

This checker checks for these issues:

- **Pointer or reference to stack variable leaving scope**.
- **Use of automatic variable as putenv-family function argument**.

## Examples

### Pointer or reference to stack variable leaving scope

#### Issue

**Pointer or reference to stack variable leaving scope** occurs when a pointer or reference to a local variable leaves the scope of the variable. For instance:

- A function returns a pointer to a local variable.
- A function performs the assignment `globPtr = &locVar`. `globPtr` is a global pointer variable and `locVar` is a local variable.
- A function performs the assignment `*paramPtr = &locVar`. `paramPtr` is a function parameter that is, for instance, an `int**` pointer and `locVar` is a local `int` variable.
- A C++ method performs the assignment `memPtr = &locVar`. `memPtr` is a pointer data member of the class the method belongs to. `locVar` is a variable local to the method.

The defect also applies to memory allocated using the `alloca` function. The defect does not apply to static, local variables. Polyspace assumes that the local objects within a function definition are in the same scope.

#### Risk

Local variables are allocated an address on the stack. Once the scope of a local variable ends, this address is available for reuse. Using this address to access the local variable value outside the variable scope can cause unexpected behavior.

If a pointer to a local variable leaves the scope of the variable, Polyspace Bug Finder highlights the defect. The defect appears even if you do not use the address stored in the pointer. For maintainable code, it is a good practice to not allow the pointer to leave the variable scope. Even if you do not use the address in the pointer now, someone else using your function can use the address, causing undefined behavior.

#### Fix

Do not allow a pointer or reference to a local variable to leave the variable scope.

#### Example - Pointer to Local Variable Returned from Function

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In this example, `func1` returns a pointer to local variable `ret`.

In `main`, `ptr` points to the address of the local variable. When `ptr` is accessed in `func2`, the access is illegal because the scope of `ret` is limited to `func1`,

### Use of automatic variable as putenv-family function argument

#### Issue

**Use of automatic variable as `putenv`-family function argument** occurs when the argument of a `putenv`-family function is a local variable with automatic duration.

#### Risk

The function `putenv(char *string)` inserts a pointer to its supplied argument into the environment array, instead of making a copy of the argument. If the argument is an automatic variable, its memory can be overwritten after the function containing the `putenv()` call returns. A subsequent call to `getenv()` from another function returns the address of an out-of-scope variable that cannot be dereferenced legally. This out-of-scope variable can cause environment variables to take on unexpected values, cause the program to stop responding, or allow arbitrary code execution vulnerabilities.

#### Fix

Use `setenv()`/`unsetenv()` to set and unset environment variables. Alternatively, use `putenv`-family function arguments with dynamically allocated memory, or, if your application has no reentrancy requirements, arguments with static duration. For example, a single thread execution with no recursion or interrupts does not require reentrancy. It cannot be called (reentered) during its execution.

#### Example - Automatic Variable as Argument of `putenv()`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
```

```
{
    char env[SIZE1024];
    int retval = sprintf(env, "TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
        /* Handle error */
    }
    /* Environment variable TEST is set using putenv().
    The argument passed to putenv is an automatic variable. */
    retval = putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
}
```

In this example, `sprintf()` stores the character string `TEST=var` in `env`. The value of the environment variable `TEST` is then set to `var` by using `putenv()`. Because `env` is an automatic variable, the value of `TEST` can change once `func()` returns.

**Correction — Use `static` Variable for Argument of `putenv()`**

Declare `env` as a static-duration variable. The memory location of `env` is not overwritten for the duration of the program, even after `func()` returns.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024
void func(int var)
{
    /* static duration variable */
    static char env[SIZE1024];
    int retval = sprintf(env,"TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
        /* Handle error */
    }

    /* Environment variable TEST is set using putenv() */
    retval=putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
}
```

**Correction — Use `setenv()` to Set Environment Variable Value**

To set the value of `TEST` to `var`, use `setenv()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    /* Environment variable TEST is set using setenv() */
    int retval = setenv("TEST", var ? "1" : "0", 1);

    if (retval != 0) {
        /* Handle error */
    }
}
```

### Check Information

**Decidability:** Undecidable

## Version History

**Introduced in R2019a**

### See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

#### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [alignconv]

Converting pointer values to more strictly aligned pointer types

## Description

### Rule Definition

*Converting pointer values to more strictly aligned pointer types.*

### Polyspace Implementation

This checker checks for **Wrong allocated object size for cast**.

## Examples

### Wrong allocated object size for cast

#### Issue

**Wrong allocated object size for cast** occurs during pointer conversion when the pointer's address is misaligned. If a pointer is converted to a different pointer type, the size of the allocated memory must be a multiple of the size of the destination pointer.

#### Risk

Dereferencing a misaligned pointer has undefined behavior and can cause your program to crash.

#### Fix

Suppose you convert a pointer `ptr1` to `ptr2`. If `ptr1` points to a buffer of `N` bytes and `ptr2` is a *`type`* `*` pointer where `sizeof(`*`type`*`)` is `n` bytes, make sure that `N` is an integer multiple of `n`.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Dynamic Allocation of Pointers

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(13);
    long *dest;

    dest = (long*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to a `long*`. The dynamically allocated memory of `ptr`, 13 bytes, is not a multiple of the size of `dest`, 4 bytes. This misalignment causes the **Wrong allocated object size for cast** defect.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the allocated memory to 12 instead of 13.

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(12);
    long *dest;

    dest = (long*)ptr;
}
```

#### Example - Static Allocation of Pointers

```
void static_non_align(void){
    char arr[13], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to an `int*` in line 6. `ptr` has a memory size of 13 bytes because the array `arr` has a size of 13 bytes. The size of `dest` is 4 bytes, which is not a multiple of 13. This misalignment causes the **Wrong allocated object size for cast** defect.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the size of the array `arr` to a multiple of 4.

```
void static_non_align(void){
    char arr[12], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr;
}
```

#### Example - Allocation with a Function

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;
    dest1 = (int*)my_alloc(13);  //defect
    dest2 = (char*)my_alloc(13); //not a defect
}
```

In this example, the software raises a defect on the conversion of the pointer returned by `my_alloc(13)` to an `int*` in line 11. `my_alloc(13)` returns a pointer with a dynamically allocated size of 13 bytes. The size of `dest1` is 4 bytes, which is not a divisor of 13. This misalignment causes the **Wrong allocated object size for cast** defect. In line 12, the same function call, `my_alloc(13)`, does not call a defect for the conversion to `dest2` because the size of `char*`, 1 byte, a divisor of 13.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the argument for `my_alloc` to a multiple of 4.

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(12);
    dest2 = (char*)my_alloc(13);
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

#### Topics

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [argcomp]

Calling functions with incorrect arguments

## Description

### Rule Definition

*Calling functions with incorrect arguments.*

### Polyspace Implementation

This checker checks for these issues:

- **Conflicting declarations or conflicting declaration and definition**.
- **Unreliable cast of function pointer**.

## Examples

### Conflicting declarations or conflicting declaration and definition

#### Issue

The issue occurs when all declarations of an object or function do not use the same names and type qualifiers.

The rule checker detects situations where parameter names or data types are different between multiple declarations or the declaration and the definition. The checker considers declarations in all translation units and flags issues that are not likely to be detected by a compiler.

*The checker does not flag this issue in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.

#### Risk

Consistently using parameter names and types across declarations of the same object or function encourages stronger typing. It is easier to check that the same function interface is used across all declarations.

#### Example - Mismatch in Parameter Names

```
extern int div (int num, int den);

int div(int den, int num) { /* Non compliant */
    return(num/den);
}
```

In this example, the rule is violated because the parameter names in the declaration and definition are switched.

#### Example - Mismatch in Parameter Data Types

```
typedef unsigned short width;
typedef unsigned short height;
typedef unsigned int area;

extern area calculate(width w, height h);

area calculate(width w, width h) { /* Noncompliant */
    return w*h;
}
```

In this example, the rule is violated because the second argument of the `calculate` function has data type:

- `height` in the declaration.
- `width` in the definition.

The rule is violated even though the underlying type of `height` and `width` are identical.

### Unreliable cast of function pointer

#### Issue

**Unreliable cast of function pointer** occurs when a function pointer is cast to another function pointer that has different argument or return type.

This defect applies only if the code language for the project is C.

#### Risk

If you cast a function pointer to another function pointer with different argument or return type and then use the latter function pointer to call a function, the behavior is undefined.

#### Fix

Avoid a cast between two function pointers with mismatch in argument or return types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Unreliable cast of function pointer error

```
#include <stdio.h>
#include <math.h>
#include <stdio.h>
#define PI 3.142

double Calculate_Sum(int (*fptr)(double))
{
    double  sum = 0.0;
    double  y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;

    fp = sin;
    sum = Calculate_Sum(fp);
    /* Defect: fp implicitly cast to int(*) (double) */

    printf("sum(sin): %f\n", sum);
    return 0;
}
```

The function pointer `fp` is declared as `double (*)(double)`. However in passing it to function `Calculate_Sum`, `fp` is implicitly cast to `int (*)(double)`.

#### Correction — Avoid Function Pointer Cast

One possible correction is to check that the function pointer in the definition of `Calculate_Sum` has the same argument and return type as `fp`. This step makes sure that `fp` is not implicitly cast to a different argument or return type.

```
#include <stdio.h>
#include <math.h>
#include <stdio.h>
# define PI 3.142

/*Fix: fptr has same argument and return type everywhere*/
double Calculate_Sum(double (*fptr)(double))
{
    double  sum = 0.0;
    double y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;


    fp = sin;
    sum = Calculate_Sum(fp);
    printf("sum(sin): %f\n", sum);
    return 0;
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [asyncsig]

Calling functions in the C Standard Library other than abort, _Exit, and signal from within a signal handler

## Description

### Rule Definition

*Calling functions in the C Standard Library other than abort, _Exit, and signal from within a signal handler.*

### Polyspace Implementation

This checker checks for these issues:

- **Function called from signal handler not asynchronous-safe (strict)**.
- **Function called from signal handler not asynchronous-safe**.

## Examples

### Function called from signal handler not asynchronous-safe (strict)

#### Issue

**Function called from signal handler not asynchronous-safe (strict)** occurs when a signal handler calls a function that is not asynchronous-safe according to the C standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

When you select the checker **Function called from signal handler not asynchronous-safe**, the checker detects calls to functions that are not asynchronous-safe according to the POSIX standard. **Function called from signal handler not asynchronous-safe (strict)** does not raise a defect for these cases. **Function called from signal handler not asynchronous-safe (strict)** raises a defect for functions that are asynchronous-safe according to the POSIX standard but not according to the C standard.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

#### Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

#### Fix

The C standard defines the following functions as asynchronous-safe. You can call these functions from a signal handler:

- abort()
- _Exit()
- quick_exit()
- signal()

**Example - Call to raise() Inside Signal Handler**

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}

void sig_handler(int signum)
{
    int s0 = signum;
    /* Call raise() */
    if (raise(SIGTERM) != 0) {
        /* Handle error */
    }
}

int finc(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` calls `raise()` when catching a signal. If the handler catches another signal while `raise()` is executing, the behavior of the program is undefined.

#### Correction — Remove Call to raise() in Signal Handler

According to the C standard, the only functions that you can safely call from a signal handler are `abort()`, `_Exit()`, `quick_exit()`, and `signal()`.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}
void sig_handler(int signum)
{
    int s0 = signum;


}

int func(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

### Function called from signal handler not asynchronous-safe

#### Issue

**Function called from signal handler not asynchronous-safe** occurs when a signal handler calls a function that is not asynchronous-safe according to the POSIX standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

**Risk**

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

**Fix**

The POSIX standard defines these functions as asynchronous-safe. You can call these functions from a signal handler.

| | | |
|---|---|---|
| `_exit()` | `getpgrp()` | `setsockopt()` |
| `_Exit()` | `getpid()` | `setuid()` |
| `abort()` | `getppid()` | `shutdown()` |
| `accept()` | `getsockname()` | `sigaction()` |
| `access()` | `getsockopt()` | `sigaddset()` |
| `aio_error()` | `getuid()` | `sigdelset()` |
| `aio_return()` | `kill()` | `sigemptyset()` |
| `aio_suspend()` | `link()` | `sigfillset()` |
| `alarm()` | `linkat()` | `sigismember()` |
| `bind()` | `listen()` | `signal()` |
| `cfgetispeed()` | `lseek()` | `sigpause()` |
| `cfgetospeed()` | `lstat()` | `sigpending()` |
| `cfsetispeed()` | `mkdir()` | `sigprocmask()` |
| `cfsetospeed()` | `mkdirat()` | `sigqueue()` |
| `chdir()` | `mkfifo()` | `sigset()` |
| `chmod()` | `mkfifoat()` | `sigsuspend()` |
| `chown()` | `mknod()` | `sleep()` |
| `clock_gettime()` | `mknodat()` | `sockatmark()` |
| `close()` | `open()` | `socket()` |
| `connect()` | `openat()` | `socketpair()` |
| `creat()` | `pathconf()` | `stat()` |
| `dup()` | `pause()` | `symlink()` |
| `dup2()` | `pipe()` | `symlinkat()` |
| `execl()` | `poll()` | `sysconf()` |
| `execle()` | `posix_trace_event()` | `tcdrain()` |
| `execv()` | `pselect()` | `tcflow()` |
| `execve()` | `pthread_kill()` | `tcflush()` |
| `faccessat()` | `pthread_self()` | `tcgetattr()` |
| `fchdir()` | `pthread_sigmask()` | `tcgetpgrp()` |
| `fchmod()` | `quick_exit()` | `tcsendbreak()` |

| | | |
|---|---|---|
| fchmodat() | raise() | tcsetattr() |
| fchown() | read() | tcsetpgrp() |
| fchownat() | readlink() | time() |
| fcntl() | readlinkat() | timer_getoverrun() |
| fdatasync() | recv() | timer_gettime() |
| fexecve() | recvfrom() | timer_settime() |
| fork() | recvmsg() | times() |
| fpathconf() | rename() | umask() |
| fstat() | renameat() | uname() |
| fstatat() | rmdir() | unlink() |
| fsync() | select() | unlinkat() |
| ftruncate() | sem_post() | utime() |
| futimens() | send() | utimensat() |
| getegid() | sendmsg() | utimes() |
| geteuid() | sendto() | wait() |
| getgid() | setgid() | waitpid() |
| getgroups() | setpgid() | write() |
| getpeername() | setsid() | |

Functions not in the previous table are not asynchronous-safe, and should not be called from a signal hander.

**Example - Call to `printf()` Inside Signal Handler**

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler(int signum)
{
    /* Call function printf() that is not
    asynchronous-safe */
    printf("signal %d received.", signum);
    e_flag = 1;
}

int main(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, sizeof(char));
    if (info == NULL)
    {
        /* Handle Error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

In this example, `sig_handler` calls `printf()` when catching a signal. If the handler catches another signal while `printf()` is executing, the behavior of the program is undefined.

**Correction — Set Flag Only in Signal Handler**

Use your signal handler to set only the value of a flag. `e_flag` is of type volatile `sig_atomic_t`. `sig_handler` can safely access it asynchronously.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler1(int signum)
{
    int s0 = signum;
    e_flag = 1;
}

int func(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler1) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, 1);
    if (info == NULL)
    {
        /* Handle error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

Check ISO/IEC TS 17961 (-iso-17961)

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [boolasgn]

No assignment in conditional expressions

## Description

### Rule Definition

*No assignment in conditional expressions.*

### Polyspace Implementation

This checker checks for **Invalid use of = (assignment) operator**.

## Examples

### Invalid use of = (assignment) operator

#### Issue

**Invalid use of = operator** occurs when an assignment is made inside the predicate of a conditional, such as `if` or `while`.

In C and C++, a single equal sign is an assignment not a comparison. Using a single equal sign in a conditional statement can indicate a typo or a mistake.

#### Risk

- Conditional statement tests the wrong values— The single equal sign operation assigns the value of the right operand to the left operand. Then, because this assignment is inside the predicate of a conditional, the program checks whether the new value of the left operand is nonzero or not NULL.
- Maintenance and readability issues — Even if the assignment is intended, someone reading or updating the code can misinterpret the assignment as an equality comparison instead of an assignment.

#### Fix

- If the assignment is a bug, to check for equality, add a second equal sign (==).
- If the assignment inside the conditional statement was intentional, to improve readability, separate the assignment and the test. Move the assignment outside the control statement. In the control statement, simply test the result of the assignment.

  If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

  - “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
  - “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
  - “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Single Equal Sign Inside an `if` Condition

```
#include <stdio.h>

void bad_equals_ex(int alpha, int beta)
{
    if(alpha = beta)
    {
        printf("Equal\n");
    }
}
```

The equal sign is flagged as a defect because the assignment operator is used within the predicate of the if-statement. The predicate assigns the value `beta` to `alpha`, then implicitly tests whether `alpha` is true or false.

#### Correction — Change Expression to Comparison

One possible correction is adding an additional equal sign. This correction changes the assignment to a comparison. The if condition compares whether `alpha` and `beta` are equal.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha == beta)
    {
        printf("Equal\n");
    }
}
```

#### Correction — Assignment and Comparison Inside the `if` Condition

If an assignment must be made inside the predicate, a possible correction is adding an explicit comparison. This correction assigns the value of `beta` to `alpha`, then explicitly checks whether `alpha` is nonzero. The code is clearer.

```
#include <stdio.h>

int assignment_not_zero(int alpha, int beta)
{
    if((alpha = beta) != 0)
    {
        return alpha;
    }
    else
    {
        return 0;
    }
}
```

#### Correction — Move Assignment Outside the `if` Statement

If the assignment can be made outside the control statement, one possible correction is to separate the assignment and comparison. This correction assigns the value of `beta` to `alpha` before the if. Inside the if-condition, only `alpha` is given to test if `alpha` is nonzero or not NULL.

```
#include <stdio.h>
```

```
void assign_and_print(int alpha, int beta)
{
    alpha = beta;
    if(alpha)
    {
        printf("%d", alpha);
    }
}
```

## Check Information

**Decidability:** Decidable

# Version History

**Introduced in R2019a**

## See Also

Check ISO/IEC TS 17961 (-iso-17961)

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [chreof]

Using character values that are indistinguishable from EOF

## Description

### Rule Definition

*Using character values that are indistinguishable from EOF.*

### Polyspace Implementation

This checker checks for **Character value absorbed into EOF**.

## Examples

### Character value absorbed into EOF

#### Issue

**Character value absorbed into EOF** occurs when you perform a data type conversion that makes a valid character value indistinguishable from `EOF` (End-of-File). Bug Finder flags the defect in one of the following situations:

- *End-of-File*: You perform a data type conversion such as from `int` to `char` that converts a non-EOF character value into `EOF`.

  ```
  char ch = (char)getchar();
  ```

  You then compare the result with EOF.

  ```
  if((int)ch == EOF){//...}
  ```

  The conversion can be explicit or implicit.
- *Wide End-of-File*: You perform a data type conversion that can convert a non-WEOF wide character value into WEOF, and then compare the result with WEOF.

#### Risk

The data type `char` cannot hold the value `EOF` that indicates the end of a file. Functions such as `getchar` have return type `int` to accommodate `EOF`. If you convert from `int` to `char`, the values `UCHAR_MAX` (a valid character value) and `EOF` get converted to the same value -1 and become indistinguishable from each other. When you compare the result of this conversion with `EOF`, the comparison can lead to false detection of `EOF`. This rationale also applies to wide character values and `WEOF`.

#### Fix

Perform the comparison with `EOF` or `WEOF` before conversion.

#### Example - Return Value of `getchar` Converted to `char`

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define fatal_error() abort()

char func(void)
{
    char ch;
    ch = getchar();
    if (EOF == (int)ch) {
        fatal_error();
    }
    return ch;
}
```

In this example, the return value of `getchar` is implicitly converted to `char`. If `getchar` returns `UCHAR_MAX`, it is converted to -1, which is indistinguishable from EOF. When you compare with EOF later, it can lead to a false positive.

#### Correction — Perform Comparison with EOF Before Conversion

One possible correction is to first perform the comparison with EOF, and then convert from `int` to `char`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

char func(void)
{
    int i;
    i = getchar();
    if (EOF == i) {
        fatal_error();
    }
    else {
        return (char)i;
    }
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [chrsgnext]

Passing arguments to character handling functions that are not representable as unsigned char

## Description

### Rule Definition

*Passing arguments to character handling functions that are not representable as unsigned char.*

### Polyspace Implementation

This checker checks for **Invalid use of standard library integer routine**.

## Examples

### Invalid use of standard library integer routine

#### Issue

**Invalid use of standard library integer routine** occurs when you use invalid arguments with an integer function from the standard library. This defect picks up:

- Character Conversion

  `toupper, tolower`
- Character Checks

  `isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit`
- Integer Division

  `div, ldiv`
- Absolute Values

  `abs, labs`

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.

- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Absolute Value of Large Negative

```
#include <limits.h>
#include <stdlib.h>

int absoluteValue(void) {

    int neg = INT_MIN;
    return abs(neg);
}
```

The input value to `abs` is `INT_MIN`. The absolute value of `INT_MIN` is `INT_MAX+1`. This number cannot be represented by the type `int`.

#### Correction — Change Input Argument

One possible correction is to change the input value to fit returned data type. In this example, change the input value to `INT_MIN+1`.

```
#include <limits.h>
#include <stdlib.h>

int absoluteValue(void) {

    int neg = INT_MIN+1;
    return abs(neg);
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [dblfree]

Freeing memory multiple times

## Description

### Rule Definition

*Freeing memory multiple times.*

### Polyspace Implementation

This checker checks for **Deallocation of previously deallocated pointer**.

## Examples

### Deallocation of previously deallocated pointer

#### Issue

**Deallocation of previously deallocated pointer** occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

#### Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to free this block of memory can result in a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. See if you intended to allocate a memory block to the pointer between the first deallocation and the second. Otherwise, remove the second `free` statement.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before freeing pointers, check them for NULL values and handle the error. In this way, you are protected against freeing an already freed block.

#### Example - Deallocation of Previously Deallocated Pointer Error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

#### Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
 }
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

**Topics**

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [diverr]

Integer division errors

## Description

### Rule Definition

*Integer division errors.*

### Polyspace Implementation

This checker checks for **Integer division by zero**.

### Extend Checker

When the input values are unknown and only a subset of inputs cause an issue, Bug Finder might not detect an **Integer division by zero**. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Integer division by zero

#### Issue

**Integer division by zero** occurs when the denominator of a division or modulo operation can be a zero-valued integer.

#### Risk

A division by zero can result in a program crash.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the denominator variable acquires a zero value. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

#### Correction — Check Before Division

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

#### Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2;
    int result = 0;

    result = num/denom;

    return result;
}
```

#### Example - Modulo Operation with Zero

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        arr[i] = input % i;
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

In this example, Polyspace flags the modulo operation as a division by zero. Because modulo is inherently a division operation, the divisor (right hand argument) cannot be zero. The modulo operation uses the `for` loop index as the divisor. However, the `for` loop starts at zero, which cannot be an iterator.

#### Correction — Check Divisor Before Operation

One possible correction is checking the divisor before the modulo operation. In this example, see if the index `i` is zero before the modulo operation.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        if(i != 0)
        {
             arr[i] = input % i;
        }
        else
        {
             arr[i] = input;
        }
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

#### Correction — Change Divisor

Another possible correction is changing the divisor to a nonzero integer. In this example, add one to the index before the `%` operation to avoid dividing by zero.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
         arr[i] = input % (i+1);
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [fileclose]

Failing to close files or free dynamic memory when they are no longer needed

## Description

### Rule Definition

*Failing to close files or free dynamic memory when they are no longer needed.*

### Polyspace Implementation

This checker checks for these issues:

- **Memory leak**.
- **Resource leak**.
- **Thread-specific memory leak**.

## Examples

### Memory leak

#### Issue

**Memory leak** occurs when you do not free a block of memory allocated through `malloc`, `calloc`, `realloc`, or `new`. If the memory is allocated in a function, the defect does not occur if:

- Within the function, you free the memory using `free` or `delete`.
- The function returns the pointer assigned by `malloc`, `calloc`, `realloc`, or `new`.
- The function stores the pointer in a global variable or in a parameter.

#### Risk

Dynamic memory allocation functions such as `malloc` allocate memory on the heap. If you do not release the memory after use, you reduce the amount of memory available for another allocation. On embedded systems with limited memory, you might end up exhausting available heap memory even during program execution.

#### Fix

Determine the scope where the dynamically allocated memory is accessed. Free the memory block at the end of this scope.

To free a block of memory, use the `free` function on the pointer that was used during memory allocation. For instance:

```
ptr = (int*)malloc(sizeof(int));
//...
free(ptr);
```

It is a good practice to allocate and free memory in the same module at the same level of abstraction. For instance, in this example, `func` allocates and frees memory at the same level but `func2` does not.

```
void func() {
  ptr = (int*)malloc(sizeof(int));
  {
    ...
  }
  free(ptr);
}

void func2() {
  {
   ptr = (int*)malloc(sizeof(int));
   ...
  }
  free(ptr);
}
```

See CERT-C Rule MEM00-C.

**Example - Dynamic Memory Not Released Before End of Function**

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
}
```

In this example, `pi` is dynamically allocated by `malloc`. The function `assign_memory` does not free the memory, nor does it return `pi`.

**Correction — Free Memory**

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
    free(pi);
}
```

#### Correction — Return Pointer from Dynamic Allocation

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

```
#include<stdlib.h>
#include<stdio.h>

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
            printf("Memory allocation failed");
            return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

### Resource leak

#### Issue

**Resource leak** occurs when you open a file stream by using a `FILE` pointer but do not close it before:

- The end of the pointer’s scope.
- Assigning the pointer to another stream.

#### Risk

If you do not release file handles explicitly as soon as possible, a failure can occur due to exhaustion of resources.

#### Fix

Close a `FILE` pointer before the end of its scope, or before you assign the pointer to another stream.

#### Example - FILE Pointer Not Released Before End of Scope

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`.

#### Correction — Release FILE Pointer

One possible correction is to explicitly dissociate `fp1` from the file stream of `data1.txt`.

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );
    fclose(fp1);

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

### Thread-specific memory leak

#### Issue

**Thread-specific memory leak** occurs when you do not free thread-specific dynamically allocated memory before the end of a thread.

To create thread-specific storage, you generally do these steps:

1. You create a key for thread-specific storage.
2. You create the threads.
3. In each thread, you allocate storage dynamically and then associate the key with this storage.

   After the association, you can read the stored data later using the key.
4. Before the end of the thread, you free the thread-specific memory using the key.

The checker flags execution paths in the thread where the last step is missing.

The checker works on these families of functions:

- `tss_get` and `tss_set` (C11)
- `pthread_getspecific` and `pthread_setspecific` (POSIX)

#### Risk

The data stored in the memory is available to other processes even after the threads end (memory leak). Besides security vulnerabilities, memory leaks can shrink the amount of available memory and reduce performance.

#### Fix

Free dynamically allocated memory before the end of a thread.

You can explicitly free dynamically allocated memory with functions such as `free`.

Alternatively, when you create a key, you can associate a destructor function with the key. The destructor function is called with the key value as argument at the end of a thread. In the body of the destructor function, you can free any memory associated with the key. If you use this method, Bug Finder still flags a defect. Ignore this defect with appropriate comments. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Memory Not Freed at End of Thread**

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;

  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */
  }
  print_data();
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }
  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

In this example, the start function of each thread `func` calls two functions:

- `add_data`: This function allocates storage dynamically and associates the storage with a key using the `tss_set` function.
- `print_data`: This function reads the stored data using the `tss_get` function.

At the points where `func` returns, the dynamically allocated storage has not been freed.

**Correction — Free Dynamically Allocated Memory Explicitly**

One possible correction is to free dynamically allocated memory explicitly before leaving the start function of a thread. See the highlighted change in the corrected version.

In this corrected version, a defect still appears on the `return` statement in the error handling section of `func`. The defect cannot occur in practice because the error handling section is entered only if dynamic memory allocation fails. Ignore this remaining defect with appropriate comments. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;
  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */
  }
  print_data();
  free(tss_get(key));
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }

  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [filecpy]

Copying a FILE object

## Description

### Rule Definition

*Copying a FILE object.*

### Polyspace Implementation

This checker checks for **Dereferencing a FILE* pointer**.

## Examples

### Dereferencing a FILE* pointer

#### Issue

The issue occurs when a pointer to a FILE object is dereferenced.

#### Risk

The Standard states that the address of a FILE object used to control a stream can be significant. Copying that object might not give the same behavior. This rule ensures that you cannot perform such a copy.

Directly manipulating a FILE object might be incompatible with its use as a stream designator.

#### Example - FILE* Pointer Dereferenced

```
#include <stdio.h>

void func(void) {
    FILE *pf1;
    FILE *pf2;
    FILE f3;

    pf2 = pf1;        /* Compliant */
    f3 = *pf2;        /* Non-compliant */
    pf2->_flags=0;    /* Non-compliant */
 }
```

In this example, the rule is violated when the FILE* pointer pf2 is dereferenced.

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [funcdecl]

Declaring the same function or object in incompatible ways

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*Declaring the same function or object in incompatible ways.*

### Polyspace Implementation

This checker checks for these issues:

- **Indistinguishable external identifier names**.
- **Declaration mismatch**.

## Examples

### Indistinguishable external identifier names

#### Issue

The issue occurs when external identifiers are not distinct.

#### Risk

External identifiers are ones declared with global scope or storage class `extern`.

Polyspace considers two names as distinct if there is a difference between their first 31 characters. If the difference between two names occurs only beyond the first 31 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 6 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`.

#### Example - C90: First Six Characters of Identifiers Not Unique

```
int engine_temperature_raw;
int engine_temperature_scaled;   /* Non-compliant */
int engin2_temperature;          /* Compliant */
```

In this example, the identifier `engine_temperature_scaled` has the same first six characters as a previous identifier, `engine_temperature_raw`.

#### Example - C99: First 31 Characters of Identifiers Not Unique

```
int engine_exhaust_gas_temperature_raw;
int engine_exhaust_gas_temperature_scaled; /* Non-compliant */

int eng_exhaust_gas_temp_raw;
int eng_exhaust_gas_temp_scaled;           /* Compliant */
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same first 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

#### Example - C90: First Six Characters Identifiers in Different Translation Units Differ in Case Alone

```
/* file1.c */
int abc = 0;

/* file2.c */
int ABC = 0; /* Non-compliant */
```

In this example, the implementation supports 6 significant case-insensitive characters in *external identifiers*. The identifiers in the two translation are different but are not distinct in their significant characters.

### Declaration mismatch

#### Issue

**Declaration mismatch** occurs when a function or variable declaration does not match other instances of the function or variable.

#### Risk

When a mismatch occurs between two variable declarations in different compilation units, a typical linker follows an algorithm to pick one declaration for the variable. If you expect a variable declaration that is different from the one chosen by the linker, you can see unexpected results when the variable is used.

A similar issue can occur with mismatch in function declarations.

#### Fix

The fix depends on the type of declaration mismatch. If both declarations indeed refer to the same object, use the same declaration. If the declarations refer to different objects, change the names of the one of the variables. If you change a variable name, remember to make the change in all places that use the variable.

Sometimes, declaration mismatches can occur because the declarations are affected by previous preprocessing directives. For instance, a declaration occurs in a macro, and the macro is defined on one inclusion path but undefined in another. These declaration mismatches can be tricky to debug. Identify the divergence between the two inclusion paths and fix the conflicting macro definitions.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Inconsistent Declarations in Two Files

*file1.c*

```
int foo(void) {
    return 1;
}
```

*file2.c*

```
double foo(void);

int bar(void) {
    return (int)foo();
}
```

In this example, *file1.c* declares `foo()` as returning an integer. In *file2.c*, `foo()` is declared as returning a double. This difference raises a defect on the second instance of `foo` in *file2*.

**Correction — Align the Function Return Values**

One possible correction is to change the function declarations so that they match. In this example, by changing the declaration of `foo` in *file2.c* to match *file1.c*, the defect is fixed.

*file1.c*

```
int foo(void) {
    return 1;
}
```

*file2.c*

```
int foo(void);

int bar(void) {
    return foo();
}
```

**Example - Inconsistent Structure Alignment**

| | |
|---|---|
| *test1.c*<br><br>`#include "square.h"`<br>`#include "circle.h"`<br>`struct aCircle circle;`<br>`struct aSquare square;`<br><br>`int main(){`<br>`    square.side=1;`<br>`    circle.radius=1;`<br>`    return 0;`<br>`}` | *test2.c*<br><br>`#include "circle.h"`<br>`#include "square.h"`<br>`struct aCircle circle;`<br>`struct aSquare square;`<br><br>`int main(){`<br>`    square.side=1;`<br>`    circle.radius=1;`<br>`    return 0;`<br>`}` |
| *circle.h*<br><br>`#pragma pack(1)`<br><br>`extern struct aCircle{`<br>`    int radius;`<br>`} circle;` | *square.h*<br><br>`extern struct aSquare {`<br>`    unsigned int side:1;`<br>`} square;` |

In this example, a declaration mismatch defect is raised on `square` in *square.h* because Polyspace infers that `square` in *square.h* does not have the same alignment as `square` in *test2.c*. This error occurs because the `#pragma pack(1)` statement in *circle.h* declares specific alignment. In *test2.c*, *circle.h* is included before *square.h*. Therefore, the `#pragma pack(1)` statement from *circle.h* is not reset to the default alignment after the `aCircle` structure. Because of this omission, *test2.c* infers that the `aSquare square` structure also has an alignment of 1 byte.

**Correction — Close Packing Statements**

One possible correction is to reset the structure alignment after the `aCircle` struct declaration. For the GNU or Microsoft Visual compilers, fix the defect by adding a `#pragma pack()` statement at the end of *circle.h*.

*test1.c*

```
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

*test2.c*

```
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

*circle.h*

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;

#pragma pack()
```

*square.h*

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

Other compilers require different `#pragma pack` syntax. For your syntax, see the documentation for your compiler.

**Correction — Use the `Ignore pragma pack directives` Option**

One possible correction is to add the `Ignore pragma pack directives` option to your Bug Finder analysis. If you want the structure alignment to change for each structure, and you do not want to see this **Declaration mismatch** defect, use this correction.

1. On the Configuration pane, select the **Advanced Settings** pane.
2. In the **Other** box, enter `-ignore-pragma-pack`.
3. Rerun your analysis.

   The **Declaration mismatch** defect is resolved.

## Check Information

**Decidability:** Decidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [insufmem]

Allocating insufficient memory

## Description

### Rule Definition

*Allocating insufficient memory.*

### Polyspace Implementation

This checker checks for these issues:

- **Wrong allocated object size for cast**.
- **Pointer access out of bounds**.
- **Wrong type used in sizeof**.
- **Possible misuse of sizeof**.

## Examples

### Wrong allocated object size for cast

#### Issue

**Wrong allocated object size for cast** occurs during pointer conversion when the pointer's address is misaligned. If a pointer is converted to a different pointer type, the size of the allocated memory must be a multiple of the size of the destination pointer.

#### Risk

Dereferencing a misaligned pointer has undefined behavior and can cause your program to crash.

#### Fix

Suppose you convert a pointer `ptr1` to `ptr2`. If `ptr1` points to a buffer of `N` bytes and `ptr2` is a *`type`* `*` pointer where `sizeof(`*`type`*`)` is `n` bytes, make sure that `N` is an integer multiple of `n`.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Dynamic Allocation of Pointers

```
#include <stdlib.h>
```

```
void dyn_non_align(void){
    void *ptr = malloc(13);
    long *dest;

    dest = (long*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to a `long*`. The dynamically allocated memory of `ptr`, 13 bytes, is not a multiple of the size of `dest`, 4 bytes. This misalignment causes the **Wrong allocated object size for cast** defect.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the allocated memory to 12 instead of 13.

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(12);
    long *dest;

    dest = (long*)ptr;
}
```

#### Example - Static Allocation of Pointers

```
void static_non_align(void){
    char arr[13], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to an `int*` in line 6. `ptr` has a memory size of 13 bytes because the array `arr` has a size of 13 bytes. The size of `dest` is 4 bytes, which is not a multiple of 13. This misalignment causes the **Wrong allocated object size for cast** defect.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the size of the array `arr` to a multiple of 4.

```
void static_non_align(void){
    char arr[12], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr;
}
```

#### Example - Allocation with a Function

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(13);  //defect
    dest2 = (char*)my_alloc(13); //not a defect
}
```

In this example, the software raises a defect on the conversion of the pointer returned by `my_alloc(13)` to an `int*` in line 11. `my_alloc(13)` returns a pointer with a dynamically allocated size of 13 bytes. The size of `dest1` is 4 bytes, which is not a divisor of 13. This misalignment causes the **Wrong allocated object size for cast** defect. In line 12, the same function call, `my_alloc(13)`, does not call a defect for the conversion to `dest2` because the size of `char*`, 1 byte, a divisor of 13.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the argument for `my_alloc` to a multiple of 4.

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(12);
    dest2 = (char*)my_alloc(13);
}
```

### Pointer access out of bounds

#### Issue

**Pointer access out of bounds** occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

#### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

#### Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
```

```
        }

  return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

### Wrong type used in sizeof

#### Issue

**Wrong type used in sizeof** occurs when both of the following conditions hold:

- You assign the address of a block of memory to a pointer, or transfer data between two blocks of memory. The assignment or copy uses the `sizeof` operator.

  For instance, you initialize a pointer using `malloc(sizeof(`*`type`*`))` or copy data between two addresses using `memcpy(`*`destination_ptr`*`, `*`source_ptr`*`, sizeof(`*`type`*`))`.
- You use an incorrect type as argument of the `sizeof` operator. You use the pointer type instead of the type that the pointer points to.

  For instance, to initialize a *`type*`* pointer, you use `malloc(sizeof(`*`type*`*`))` instead of `malloc(sizeof(`*`type`*`))`.

#### Risk

Irrespective of what *`type`* stands for, the expression `sizeof(`*`type*`*`)` always returns a fixed size. The size returned is the pointer size on your platform in bytes. The appearance of `sizeof(`*`type*`*`)` often indicates an unintended usage. The error can cause allocation of a memory block that is much smaller than what you need and lead to weaknesses such as buffer overflows.

For instance, assume that `structType` is a structure with ten `int` variables. If you initialize a `structType*` pointer using `malloc(sizeof(structType*))` on a 32-bit platform, the pointer is assigned a memory block of four bytes. However, to be allocated completely for one `structType` variable, the `structType*` pointer must point to a memory block of `sizeof(structType) = 10 * sizeof(int)` bytes. The required size is much greater than the actual allocated size of four bytes.

#### Fix

To initialize a *`type*`* pointer, replace `sizeof(`*`type*`*`)` in your pointer initialization expression with `sizeof(`*`type`*`)`.

#### Example - Allocate a Char Array With `sizeof`

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char*) * 5);
    free(str);

}
```

In this example, memory is allocated for the character pointer `str` using a `malloc` of five char pointers. However, `str` is a pointer to a character, not a pointer to a character pointer. Therefore the `sizeof` argument, `char*`, is incorrect.

#### Correction — Match Pointer Type to sizeof Argument

One possible correction is to match the argument to the pointer type. In this example, `str` is a character pointer, therefore the argument must also be a character.

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char) * 5);
    free(str);

}
```

### Possible misuse of sizeof

#### Issue

**Possible misuse of sizeof** occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

#### Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

#### Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.

- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

#### Example - `sizeof` Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

#### Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

#### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [intoflow]

Overflowing signed integers

## Description

### Rule Definition

*Overflowing signed integers.*

### Polyspace Implementation

This checker checks for these issues:

- **Integer overflow**.
- **Integer constant overflow**.

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

### Extend Checker

When the input values are unknown and only a subset of inputs cause an issue, Bug Finder might not detect an **Integer overflow** or **Integer constant overflow**. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Integer overflow

#### Issue

**Integer overflow** occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different integer types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer overflows on signed integers result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

#### Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

### Integer constant overflow

#### Issue

**Integer constant overflow** occurs when you assign a compile-time constant to a signed integer variable whose data type cannot accommodate the value. An n-bit signed integer holds values in the range [-$2^{n-1}$, $2^{n-1}$-1].

For instance, c is an 8-bit signed char variable that cannot hold the value 255.

```
signed char c = 255;
```

To determine the sizes of fundamental types, Bug Finder uses your specification for Target processor type (-target).

#### Risk

The default behavior for constant overflows can vary between compilers and platforms. Retaining constant overflows can reduce the portability of your code.

Even if your compilers wraps around overflowing constants with a warning, the wrap-around behavior can be unintended and cause unexpected results.

#### Fix

Check if the constant value is what you intended. If the value is correct, use a different, possibly wider, data type for the variable.

#### Example - Overflowing Constant from Macro Expansion

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    char c1 = MAX_UNSIGNED_CHAR;
    char c2 = MAX_SIGNED_CHAR+1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow. To reproduce these defects, use analysis option Target processor type (-target) where char is signed by default.

#### Correction — Use Different Data Type

One possible correction is to use a different data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR;
    unsigned char c2 = MAX_SIGNED_CHAR+1;
}
```

## Check Information

**Decidability:** Undecidable

## Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [intptrconv]

Converting a pointer to integer or integer to pointer

## Description

### Rule Definition

*Converting a pointer to integer or integer to pointer.*

### Polyspace Implementation

This checker checks for **Conversion between pointers and integers**.

## Examples

### Conversion between pointers and integers

#### Issue

The issue occurs when a conversion is performed between a pointer to object and an integer type.

Casts or implicit conversions from NULL or (void*)0 do not generate a warning.

#### Risk

Conversion between integers and pointers can cause errors or undefined behavior.

- If an integer is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.
- If a pointer is cast to an integer, the resulting value can be outside the allowed range for the integer type.

#### Example - Casts between pointer and integer

```
#include <stdbool.h>

typedef unsigned char      uint8_t;
typedef          char      char_t;
typedef unsigned short     uint16_t;
typedef signed   int       int32_t;

typedef _Bool bool_t;
uint8_t *PORTA = (uint8_t *) 0x0002;            /* Non-compliant */

void foo(void) {

    char_t c = 1;
    char_t *pc = &c;                              /* Compliant */


    uint16_t ui16   = 7U;
    uint16_t *pui16 = &ui16;                      /* Compliant */
    pui16 = (uint16_t *) ui16;                    /* Non-compliant */
    uint16_t *p;
    int32_t addr = (int32_t) p;                  /* Non-compliant */
    bool_t b = (bool_t) p;                       /* Non-compliant */
    enum etag { A, B } e = ( enum etag ) p;      /* Non-compliant */
}
```

In this example, the rule is violated when:

- The integer `0x0002` is cast to a pointer.

  If the integer defines an absolute address, it is more common to assign the address to a pointer in a header file. To avoid the assignment being flagged, you can then exclude headers files from coding rules checking. For more information, see `Do not generate results for (-do-not-generate-results-for)`.
- The pointer `p` is cast to integer types such as `int32_t`, `bool_t` or `enum etag`.

The rule is not violated when the address `&ui16` is assigned to a pointer.

### Check Information

**Decidability:** Undecidable

## Version History

**Introduced in R2019a**

### See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

#### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [inverrno]

Incorrectly setting and using errno

## Description

### Rule Definition

*Incorrectly setting and using errno.*

### Polyspace Implementation

This checker checks for these issues:

- **Misuse of errno**.
- **Errno not checked**.
- **Errno not reset**.

## Examples

### Misuse of errno

#### Issue

**Misuse of errno** occurs when you check `errno` for error conditions in situations where checking `errno` does not guarantee the absence of errors. In some cases, checking `errno` can lead to false positives.

For instance, you check `errno` following calls to the functions:

- `fopen`: If you follow the ISO Standard, the function might not set `errno` on errors.
- `atof`: If you follow the ISO Standard, the function does not set `errno`.
- `signal`: The `errno` value indicates an error only if the function returns the `SIG_ERR` error indicator.

#### Risk

The ISO C Standard does not enforce that these functions set `errno` on errors. Whether the functions set `errno` or not is implementation-dependent.

To detect errors, if you check `errno` alone, the validity of this check also becomes implementation-dependent.

In some cases, the `errno` value indicates an error only if the function returns a specific error indicator. If you check `errno` before checking the function return value, you can see false positives.

#### Fix

For information on how to detect errors, see the documentation for that specific function.

Typically, the functions return an out-of-band error indicator to indicate errors. For instance:

- fopen returns a null pointer if an error occurs.
- signal returns the SIG_ERR error indicator and sets errno to a positive value. Check errno only after you have checked the function return value.

**Example - Incorrectly Checking for errno After fopen Call**

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    errno = 0;
    fileptr = fopen(temp_filename, "w+b");
    if (errno != 0) {
        if (fileptr != NULL) {
            (void)fclose(fileptr);
        }
        /* Handle error */
        fatal_error();
    }
    return fileptr;
}
```

In this example, errno is the first variable that is checked after a call to fopen. You might expect that fopen changes errno to a nonzero value if an error occurs. If you run this code with an implementation of fopen that does not set errno on errors, you might miss an error condition. In this situation, fopen can return a null pointer that escapes detection.

**Correction — Check Return Value of fopen After Call**

One possible correction is to only check the return value of fopen for a null pointer.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    fileptr = fopen(temp_filename, "w+b");
    if (fileptr == NULL) {
        fatal_error();
    }
    return fileptr;
}
```

### Errno not checked

#### Issue

**Errno not checked** occurs when you call a function that sets `errno` to indicate error conditions, but do not check `errno` after the call. For these functions, checking `errno` is the only reliable way to determine if an error occurred.

Functions that set `errno` on errors include:

- `fgetwc`, `strtol`, and `wcstol`.

  For a comprehensive list of functions, see documentation about errno.
- POSIX `errno`-setting functions such as `encrypt` and `setkey`.

#### Risk

To see if the function call completed without errors, check `errno` for error values.

The return values of these `errno`-setting functions do not indicate errors. The return value can be one of the following:

- `void`
- Even if an error occurs, the return value can be the same as the value from a successful call. Such return values are called in-band error indicators.

You can determine if an error occurred only by checking `errno`.

For instance, `strtol` converts a string to a long integer and returns the integer. If the result of conversion overflows, the function returns `LONG_MAX` and sets `errno` to `ERANGE`. However, the function can also return `LONG_MAX` from a successful conversion. Only by checking `errno` can you distinguish between an error and a successful conversion.

#### Fix

Before calling the function, set `errno` to zero.

After the function call, to see if an error occurred, compare `errno` to zero. Alternatively, compare `errno` to known error indicator values. For instance, `strtol` sets `errno` to `ERANGE` to indicate errors.

The error message in the Polyspace result shows the error indicator value that you can compare to.

#### Example - `errno` Not Checked After Call to `strtol`

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    long val = strtol(str, &endptr, base);
    printf("Return value of strtol() = %ld\n", val);
}
```

You are using the return value of `strtol` without checking `errno`.

#### Correction — Check `errno` After Call

Before calling `strtol`, set `errno` to zero . After a call to `strtol`, check the return value for `LONG_MIN` or `LONG_MAX` and `errno` for `ERANGE`.

```
#include<stdlib.h>
#include<stdio.h>
#include<errno.h>
#include<limits.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    errno = 0;
    long val = strtol(str, &endptr, base);
    if((val == LONG_MIN || val == LONG_MAX) && errno == ERANGE) {
         printf("strtol error");
         exit(EXIT_FAILURE);
    }
    printf("Return value of strtol() = %ld\n", val);
}
```

### Errno not reset

#### Issue

**Errno not reset** occurs when you do not reset `errno` before calling a function that sets `errno` to indicate error conditions. However, you check `errno` for those error conditions after the function call.

#### Risk

The `errno` is not clean and can contain values from a previous call. Checking `errno` for errors can give the false impression that an error occurred.

`errno` is set to zero at program startup but subsequently, `errno` is not reset by a C standard library function. You must explicitly set `errno` to zero when required.

#### Fix

Before calling a function that sets `errno` to indicate error conditions, reset `errno` to zero explicitly.

#### Example - `errno` Not Reset Before Call to `strtod`

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()
```

```
double func(const char *s1, const char *s2)
{
    double f1;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
                  {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

In this example, `errno` is not reset to 0 before the first call to `strtod`. Checking `errno` for 0 later can lead to a false positive.

**Correction — Reset `errno` Before Call**

One possible correction is to reset `errno` to 0 before calling `strtod`.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    errno = 0;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
              {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

## Check Information

**Decidability:** Undecidable

## Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [invfmtstr]

Using invalid format strings

## Description

### Rule Definition

*Using invalid format strings.*

### Polyspace Implementation

This checker checks for **Format string specifiers and arguments mismatch**.

## Examples

### Format string specifiers and arguments mismatch

#### Issue

**Format string specifiers and arguments mismatch** occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

#### Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

#### Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
const char *message = "License not available";
int err_number = ;-4
printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {
    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

#### Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

#### Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

#### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [invptr]

Forming or using out-of-bounds pointers or array subscripts

## Description

### Rule Definition

*Forming or using out-of-bounds pointers or array subscripts.*

### Polyspace Implementation

This checker checks for these issues:

- **Array access out of bounds**.
- **Pointer access out of bounds**.

### Extend Checker

When the input values are unknown and only a subset of inputs cause an issue, Bug Finder might not detect an **Array access out of bounds**. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

## Examples

### Array access out of bounds

#### Issue

**Array access out of bounds** occurs when an array index falls outside the range `[0...array_size-1]` during array access.

#### Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Array Access Out of Bounds Error**

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

**Correction — Keep Array Index Within Array Bounds**

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

### Pointer access out of bounds

#### Issue

**Pointer access out of bounds** occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

#### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

#### Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

### Check Information

**Decidability:** Undecidable

## Version History

**Introduced in R2019a**

### See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

#### Topics

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [ioileave]

Interleaving stream inputs and outputs without a flush or positioning call

## Description

### Rule Definition

*Interleaving stream inputs and outputs without a flush or positioning call.*

### Polyspace Implementation

This checker checks for **Alternating input and output from a stream without flush or positioning call**.

## Examples

### Alternating input and output from a stream without flush or positioning call

#### Issue

**Alternating input and output from a stream without flush or positioning call** occurs when:

- You do not perform a flush or function positioning call between an output operation and a following input operation on a file stream in update mode.
- You do not perform a function positioning call between an input operation and a following output operation on a file stream in update mode.

#### Risk

Alternating input and output operations on a stream without an intervening flush or positioning call is undefined behavior.

#### Fix

Call `fflush()` or a file positioning function such as `fseek()` or `fsetpos()` between output and input operations on an update stream.

Call a file positioning function between input and output operations on an update stream.

#### Example - Read After Write Without Intervening Flush

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";

void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;
    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Read operation after write without
    intervening flush. */
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
          (void)fclose(file);
          /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

In this example, the file `demo.txt` is opened for reading and appending. After the call to `fwrite()`, a call to `fread()` without an intervening flush operation is undefined behavior.

**Correction — Call `fflush()` Before the Read Operation**

After writing data to the file, before calling `fread()`, perform a flush call.

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";


void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
```

```
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Buffer flush after write and before read */
    if (fflush(file) != 0)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

Check ISO/IEC TS 17961 (-iso-17961)

### Topics

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [liberr]

Failing to detect and handle standard library errors

## Description

#### Rule Definition

*Failing to detect and handle standard library errors.*

#### Polyspace Implementation

This checker checks for these issues:

- **Returned value of a sensitive function not checked**.
- **Unprotected dynamic memory allocation**.

## Examples

### Returned value of a sensitive function not checked

#### Issue

**Returned value of a sensitive function not checked** occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `thrd_create`)
- Lock or unlock memory segments (for example, `mlock`)

#### Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

#### Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

#### Example - Sensitive Function Return Ignored

```
#include<stdio.h>
#include <wchar.h>
#include <locale.h>
void initialize(size_t n, size_t* size, wchar_t *wcs, const char *utf8) {

    scanf("%d",&n); //Noncompliant
    setlocale (LC_CTYPE, "en_US.UTF-8");   //Noncompliant
    *size = mbstowcs (wcs, utf8, n);
}
```

This example shows a call to the sensitive function `scanf()`. The return value of `scanf()` is ignored, causing a defect. Similarly, the pointer returned by `setlocale` is not checked. When `setlocal` returns a `NULL` pointer, the call to `mbstowcs` might fail or produce unexpected results. Polyspace flags these calls to sensitive functions when their returns are not checked.

#### Correction — Cast Function to (void)

One possible correction is to cast the functions to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of these sensitive functions.

```
#include<stdio.h>
#include <wchar.h>
#include <locale.h>
void initialize(size_t n, size_t* size, wchar_t *wcs, const char *utf8) {

    (void)scanf("%d",&n); //Compliant
    (void)setlocale (LC_CTYPE, "en_US.UTF-8");   //Compliant
    *size = mbstowcs (wcs, utf8, n);
}
```

#### Correction — Test Return Value

One possible correction is to test the return value of `scanf` and `setlocale` to check for errors.

```
#include<stdio.h>
#include <wchar.h>
#include <locale.h>
void initialize(size_t n, size_t* size, wchar_t *wcs, const char *utf8) {

    int flag = scanf("%d",&n);
    if(flag>0){ //Compliant
        // action
    }
    char* status = setlocale (LC_CTYPE, "en_US.UTF-8");
    if(status!=NULL){//Compliant
        *size = mbstowcs (wcs, utf8, n);
    }

}
```

#### Example - Critical Function Return Ignored

```
#include <threads.h>
int thrd_func(void);
void initialize() {
    thrd_t thr;
    int n = 1;

    (void) thrd_create(&thr,thrd_func,&n);
}
```

In this example, a critical function `thrd_create` is called and its return value is ignored by casting to void, but because `thrd_create` is a critical function, Polyspace does not ignore this *Return value of a sensitive function not checked* defect.

#### Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
 #include <threads.h>
int thrd_func(void);
void initialize() {
    thrd_t thr;
    int n = 1;
    if( thrd_success!= thrd_create(&thr,thrd_func,&n) ){
        // handle error

    }
}
```

### Unprotected dynamic memory allocation

#### Issue

**Unprotected dynamic memory allocation** occurs when you do not check after dynamic memory allocation whether the memory allocation succeeded.

#### Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

#### Fix

Check the return value of `malloc`, `calloc`, or `realloc` for NULL before accessing the allocated memory location.

```
#define SIZE 10
//...
int *ptr = malloc(SIZE * sizeof(int));
```

```
if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

**Example - Unprotected dynamic memory allocation error**

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));

  *p = 2;
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function `calloc` returns `NULL` to `p`. Before accessing the memory through `p`, the code does not check whether `p` is `NULL`

**Correction — Check for NULL Value**

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [libmod]

Modifying the string returned by getenv, localeconv, setlocale, and strerror

## Description

### Rule Definition

*Modifying the string returned by getenv, localeconv, setlocale, and strerror.*

### Polyspace Implementation

This checker checks for **Modification of internal buffer returned from nonreentrant standard function**.

## Examples

### Modification of internal buffer returned from nonreentrant standard function

#### Issue

**Modification of internal buffer returned from nonreentrant standard function** occurs when the following happens:

- A nonreentrant standard function returns a pointer.
- You attempt to write to the memory location that the pointer points to.

Nonreentrant standard functions that return a non `const`-qualified pointer to an internal buffer include `getenv`, `getlogin`, `crypt`, `setlocale`, `localeconv`, `strerror` and others.

#### Risk

Modifying the internal buffer that a nonreentrant standard function returns can cause the following issues:

- It is possible that the modification does not succeed or alters other internal data.

  For instance, `getenv` returns a pointer to an environment variable value. If you modify this value, you alter the environment of the process and corrupt other internal data.
- Even if the modification succeeds, it is possible that a subsequent call to the same standard function does not return your modified value.

  For instance, you modify the environment variable value that `getenv` returns. If another process, thread, or signal handler calls `setenv`, the modified value is overwritten. Therefore, a subsequent call to `getenv` does not return your modified value.

#### Fix

Avoid modifying the internal buffer using the pointer returned from the function.

#### Example - Modification of getenv Return Value

```
#include <stdlib.h>
#include <string.h>
```

```
void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        strncpy(env, "C", 1);
        printstr(env);
    }
}
```

In this example, the first argument of `strncpy` is the return value from a nonreentrant standard function `getenv`. The behavior can be undefined because `strncpy` modifies this argument.

**Correction - Copy Return Value of `getenv` and Modify Copy**

One possible solution is to copy the return value of `getenv` and pass the copy to the `strncpy` function.

```
#include <stdlib.h>
#include <string.h>
enum {
    SIZE20 = 20
};

void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        char env_cp[SIZE20];
        strncpy(env_cp, env, SIZE20);
        strncpy(env_cp, "C", 1);
        printstr(env_cp);
    }
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [libptr]

Forming invalid pointers by library function

## Description

### Rule Definition

*Forming invalid pointers by library function.*

### Polyspace Implementation

This checker checks for these issues:

- **Use of path manipulation function without maximum sized buffer checking**.
- **Invalid use of standard library memory routine**.
- **Invalid use of standard library string routine**.
- **Destination buffer overflow in string manipulation**.

## Examples

### Use of path manipulation function without maximum sized buffer checking

#### Issue

**Use of path manipulation function without maximum-sized buffer checking** occurs when the destination argument of a path manipulation function such as `realpath` or `getwd` has a buffer size less than `PATH_MAX` bytes.

#### Risk

A buffer smaller than `PATH_MAX` bytes can overflow but you cannot test the function return value to determine if an overflow occurred. If an overflow occurs, following the function call, the content of the buffer is undefined.

For instance, `char *getwd(char *buf)` copies an absolute path name of the current folder to its argument. If the length of the absolute path name is greater than `PATH_MAX` bytes, `getwd` returns `NULL` and the content of `*buf` is undefined. You can test the return value of `getwd` for `NULL` to see if the function call succeeded.

However, if the allowed buffer for `buf` is less than `PATH_MAX` bytes, a failure can occur for a smaller absolute path name. In this case, `getwd` does not return `NULL` even though a failure occurred. Therefore, the allowed buffer for `buf` must be `PATH_MAX` bytes long.

#### Fix

Possible fixes are:

- Use a buffer size of `PATH_MAX` bytes. If you obtain the buffer from an unknown source, before using the buffer as argument of `getwd` or `realpath` function, make sure that the size is less than `PATH_MAX` bytes.

- Use a path manipulation function that allows you to specify a buffer size.

  For instance, if you are using `getwd` to get the absolute path name of the current folder, use `char *getcwd(char *buf, size_t size);` instead. The additional argument `size` allows you to specify a size greater than or equal to `PATH_MAX`.
- Allow the function to allocate additional memory dynamically, if possible.

  For instance, `char *realpath(const char *path, char *resolved_path);` dynamically allocates memory if `resolved_path` is `NULL`. However, you have to deallocate this memory later using the `free` function.

#### Example - Possible Buffer Overflow in Use of `getwd` Function

```
#include <unistd.h>
#include <linux/limits.h>
#include <stdio.h>

void func(void) {
    char buf[PATH_MAX];
    if (getwd(buf+1)!= NULL)         {
        printf("cwd is %s\n", buf);
    }
}
```

In this example, although the array `buf` has `PATH_MAX` bytes, the argument of `getwd` is `buf + 1`, whose allowed buffer is less than `PATH_MAX` bytes.

#### Correction — Use Array of Size `PATH_MAX` Bytes

One possible correction is to use an array argument with size equal to `PATH_MAX` bytes.

```
#include <unistd.h>
#include <linux/limits.h>
#include <stdio.h>

void func(void) {
    char buf[PATH_MAX];
    if (getwd(buf)!= NULL)         {
        printf("cwd is %s\n", buf);
    }
}
```

### Invalid use of standard library memory routine

#### Issue

**Invalid use of standard library memory routine** occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

#### Risk

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do

not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Use of Standard Library Memory Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

#### Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

### Invalid use of standard library string routine

#### Issue

**Invalid use of standard library string routine** occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source);
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Use of Standard Library String Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
{
 char *res;
 char gbuffer[5],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);
 /* Error: Size of text is less than gbuffer */

 return(res);
}
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

#### Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Destination buffer overflow in string manipulation

#### Issue

**Destination buffer overflow in string manipulation** occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

#### Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [libuse]

Using an object overwritten by getenv, localeconv, setlocale, and strerror

## Description

### Rule Definition

*Using an object overwritten by getenv, localeconv, setlocale, and strerror.*

### Polyspace Implementation

This checker checks for **Misuse of return value from nonreentrant standard function**.

## Examples

### Misuse of return value from nonreentrant standard function

#### Issue

**Misuse of return value from nonreentrant standard function** occurs when these events happen in this sequence:

1. You point to the buffer returned from a nonreentrant standard function such as `getenv` or `setlocale`.

   ```
   user = getenv("USER");
   ```
2. You call that nonreentrant standard function again.

   ```
   user2 = getenv("USER2");
   ```
3. You use or dereference the pointer from the first step expecting the buffer to remain unmodified since that step. In the meantime, the call in the second step has modified the buffer.

   For instance:

   ```
   var=*user;
   ```

In some cases, the defect might appear even if you do not call the `getenv` function a second time but simply return the pointer. For instance:

```
char* func() {
     user=getenv("USER");
     .
     .
     return user;
}
```

For information on which functions are covered by this defect, see documentation on nonreentrant standard functions.

#### Risk

The C Standard allows nonreentrant functions such as `getenv` to return a pointer to a *static* buffer. Because the buffer is static, a second call to `getenv` modifies the buffer. If you continue to use the

pointer returned from the first call past the second call, you can see unexpected results. The buffer that it points to no longer has values from the first call.

The defect appears even if you do not call `getenv` a second time but simply return the pointer. The reason is that someone calling your function might use the returned pointer *after* a second call to `getenv`. By returning the pointer from your call to `getenv`, you make your function unsafe to use.

The same rationale is true for other nonreentrant functions covered by this defect.

**Fix**

After the first call to `getenv`, make a copy of the buffer that the returned pointer points to. After the second call to `getenv`, use this copy. Even if the second call modifies the buffer, your copy is untouched.

**Example - Return from getenv Used After Second Call to getenv**

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");   /* First call */
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');

        if (user_name_from_home != NULL) {
            user = getenv("USER");   /* Second call */
            if ((user != NULL) &&
                (strcmp(user, user_name_from_home) == 0))
            {
                result = 1;
            }
        }
    }
    return result;
}
```

In this example, the pointer `user_name_from_home` is derived from the pointer `home`. `home` points to the buffer returned from the first call to `getenv`. Therefore, `user_name_from_home` points to a location in the same buffer.

After the second call to `getenv`, the buffer is modified. If you continue to use `user_name_from_home`, you can get unexpected results.

**Correction — Make Copy of Buffer Before Second Call**

If you want to access the buffer from the first call to `getenv` past the second call, make a copy of the buffer after the first call. One possible correction is to use the `strdup` function to make the copy.

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');
        if (user_name_from_home != NULL) {
            /* Make copy before second call */
            char *saved_user_name_from_home = strdup(user_name_from_home);
            if (saved_user_name_from_home != NULL) {
                user = getenv("USER");
                if ((user != NULL) &&
                    (strcmp(user, saved_user_name_from_home) == 0))
                {
                    result = 1;
                }
                free(saved_user_name_from_home);
            }
        }
    }
    return result;
}
```

### Check Information

**Decidability:** Undecidable

## Version History

**Introduced in R2019a**

### See Also

Check ISO/IEC TS 17961 (-iso-17961)

**Topics**

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [nonnullcs]

Passing a non-null-terminated character sequence to a library function

## Description

### Rule Definition

*Passing a non-null-terminated character sequence to a library function.*

### Polyspace Implementation

This checker checks for **Invalid use of standard library string routine**.

## Examples

### Invalid use of standard library string routine

#### Issue

**Invalid use of standard library string routine** occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source);
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Use of Standard Library String Routine Error

```
#include <string.h>
#include <stdio.h>
```

```
 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

**Correction — Use Valid Arguments**

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

**Topics**

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [nullref]

Dereferencing an out-of-domain pointer

## Description

### Rule Definition

*Dereferencing an out-of-domain pointer.*

### Polyspace Implementation

This checker checks for these issues:

- **Unsafe pointer arithmetic**.
- **Invalid use of standard library memory routine**.
- **Null pointer**.
- **Arithmetic operation with NULL pointer**.
- **Invalid use of standard library string routine**.

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

### Extend Checker

When the input values are unknown and only a subset of inputs cause an issue, Bug Finder might not detect a **Null pointer**. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Unsafe pointer arithmetic

#### Issue

The issue occurs when a pointer resulting from arithmetic on a pointer operand does not address an element of the same array as that pointer operand.

Polyspace reports violations of this rule when your code has these issues:

- `Array access out of bounds`
- `Pointer access out of bounds`

#### Risk

Using an invalid array subscript can lead to erroneous behavior of the program. Run-time derived array subscripts are especially troublesome because they cannot be easily checked by manual review or static analysis.

The C Standard defines the creation of a pointer to one beyond the end of the array. The rule permits the C Standard. Dereferencing a pointer to one beyond the end of an array causes undefined behavior and is noncompliant.

### Invalid use of standard library memory routine

#### Issue

**Invalid use of standard library memory routine** occurs when a memory library function is called with invalid arguments. For instance, the memcpy function copies to an array that cannot accommodate the number of bytes copied.

#### Risk

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Use of Standard Library Memory Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string str2 is 5, but six characters of string str1 are copied into str2 using the memcpy function.

#### Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of str2 so that it accommodates the characters copied with the memcpy function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

### Null pointer

#### Issue

**Null pointer** occurs when you use a pointer with a value of `NULL` as if it points to a valid memory location.

#### Risk

Dereferencing a null pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

#### Fix

Check a pointer for `NULL` before dereference.

If the issue occurs despite an earlier check for `NULL`, look for intermediate events between the check and the subsequent dereference. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

#### Example - Null pointer error

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 int* p=NULL;

 *p=arr[0];
 /* Defect: Null pointer dereference */

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

The pointer p is initialized with value of NULL. However, when the value arr[0] is written to *p, p is assumed to point to a valid memory location.

#### Correction — Assign Address to Null Pointer Before Dereference

One possible correction is to initialize p with a valid memory address before dereference.

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 /* Fix: Assign address to null pointer */
 int* p=&arr[0];

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

### Arithmetic operation with NULL pointer

#### Issue

**Arithmetic operation with NULL pointer** occurs when an arithmetic operation involves a pointer whose value is NULL.

#### Risk

Performing pointer arithmetic on a null pointer and dereferencing the resulting pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

#### Fix

Check a pointer for NULL before arithmetic operations on the pointer.

If the issue occurs despite an earlier check for NULL, look for intermediate events between the check and the subsequent dereference. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

#### Example - Arithmetic Operation with NULL Pointer Error

```
#include<stdlib.h>

int Check_Next_Value(int *loc, int val)
 {
  int *ptr = loc, found = 0;

  if (ptr==NULL)
   {
      ptr++;
      /* Defect: NULL pointer shifted */
      if (*ptr==val) found=1;
   }

  return(found);
 }
```

When ptr is a NULL pointer, the code enters the if statement body. Therefore, a NULL pointer is shifted in the statement ptr++.

#### Correction — Avoid NULL Pointer Arithmetic

One possible correction is to perform the arithmetic operation when ptr is not NULL.

```
#include<stdlib.h>

int Check_Next_Value(int *loc, int val)
 {
  int *ptr = loc, found = 0;

  /* Fix: Perform operation when ptr is not NULL */
  if (ptr!=NULL)
   {
      ptr++;

      if (*ptr==val) found=1;
   }

  return(found);
 }
```

### Invalid use of standard library string routine

#### Issue

**Invalid use of standard library string routine** occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the strcpy function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the strcpy function:

```
char * strcpy(char * destination, const char* source);
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to strcpy. In some cases, you can use an alternative function to avoid the error. For instance, instead of strcpy, you can use strncpy to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Invalid Use of Standard Library String Routine Error**

```
#include <string.h>
#include <stdio.h>

char* Copy_String(void)
{
 char *res;
 char gbuffer[5],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);
 /* Error: Size of text is less than gbuffer */

 return(res);
}
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

**Correction — Use Valid Arguments**

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

**Topics**

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [padcomp]

Comparison of padding data

## Description

### Rule Definition

*Comparison of padding data.*

### Polyspace Implementation

This checker checks for **Memory comparison of padding data**.

## Examples

### Memory comparison of padding data

#### Issue

**Memory comparison of padding data** occurs when you use the memcmp function to compare two structures as a whole. In the process, you compare meaningless data stored in the structure padding.

For instance:

```
typedef struct structType {
    char member1;
    int member2;
    //...
}myStruct;

myStruct var1;
myStruct var2;
//...
if(memcmp(&var1,&var2,sizeof(var1)))//Noncompliant
{
//...
}
```

#### Risk

If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. For an example of padding, see Higher Estimate of Size of Local Variables.

The content of these extra padding bytes is meaningless. The C Standard allows the content of these bytes to be indeterminate, giving different compilers latitude to implement their own padding. If you perform a byte-by-byte comparison of structures with memcmp, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.

#### Fix

Instead of comparing two structures in one attempt, compare the structures member by member.

For efficient code, write a function that does the comparison member by member. Use this function for comparing two structures.

You can use memcmp for byte-by-byte comparison of structures only if you know that the structures do not contain padding. Typically, to prevent padding, you use specific attributes or pragmas such as #pragma pack. However, these attributes or pragmas are not supported by all compilers and make your code implementation-dependent. If your structures contain bit-fields, using these attributes or pragmas cannot prevent padding.

#### Example - Structures Compared with memcmp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{

    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    if (0 == memcmp(left, right, sizeof(S_Padding)))
    {
        return 1;
    }
    else
        return 0;
}
```

In this example, memcmp compares byte-by-byte the two structures that left and right point to. Even if the values stored in the structure members are the same, the comparison can show an inequality if the meaningless values in the padding bytes are not the same.

#### Correction — Compare Structures Member by Member

One possible correction is to compare individual structure members.

**Note** You can compare entire arrays by using memcmp. All members of an array have the same data type. Padding bytes are not required to store arrays.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{
    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    return ((left->c == right->c) &&
            (left->i == right->i) &&
            (left->bf1 == right->bf1) &&
            (left->bf2 == right->bf2) &&
            (memcmp(left->buffer, right->buffer, 20) == 0));
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

Check ISO/IEC TS 17961 (-iso-17961)

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [ptrcomp]

Accessing an object through a pointer to an incompatible type

## Description

### Rule Definition

*Accessing an object through a pointer to an incompatible type.*

### Polyspace Implementation

This checker checks for **Conversion between pointers to different objects**.

## Examples

### Conversion between pointers to different objects

#### Issue

The issue occurs when a cast is performed between a pointer to object type and a pointer to a different object type.

#### Risk

If a pointer to an object is cast into a pointer to a different object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.

Even if the conversion produces a pointer that is correctly aligned, the behavior can be undefined if the pointer is used to access an object.

Exception: You can convert a pointer to object type into a pointer to one of the following types:

- `char`
- `signed char`
- `unsigned char`

#### Example - Noncompliant: Cast to Pointer Pointing to Object of Wider Type

```
signed   char *p1;
unsigned int *p2;

void foo(void){
  p2 = ( unsigned int * ) p1;     /* Non-compliant */
}
```

In this example, `p1` can point to a `signed char` object. However, `p1` is cast to a pointer that points to an object of wider type, `unsigned int`.

#### Example - Noncompliant: Cast to Pointer Pointing to Object of Narrower Type

```
extern unsigned int read_value ( void );
extern void display ( unsigned int n );
```

```
void foo ( void ){
  unsigned int u = read_value ( );
  unsigned short *hi_p = ( unsigned short * ) &u;    /* Non-compliant  */
  *hi_p = 0;
  display ( u );
}
```

In this example, u is an unsigned int variable. &u is cast to a pointer that points to an object of narrower type, unsigned short.

On a big-endian machine, the statement *hi_p = 0 attempts to clear the high bits of the memory location that &u points to. But, from the result of display(u), you might find that the high bits have not been cleared.

**Example - Compliant: Cast Adding a Type Qualifier**

```
const short *p;
const volatile short *q;
void foo (void){
  q = ( const volatile short * ) p;  /* Compliant */
}
```

In this example, both p and q can point to short objects. The cast between them adds a volatile qualifier only and is therefore compliant.

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

Check ISO/IEC TS 17961 (-iso-17961)

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [ptrobj]

Subtracting or comparing two pointers that do not refer to the same array

## Description

### Rule Definition

*Subtracting or comparing two pointers that do not refer to the same array.*

### Polyspace Implementation

This checker checks for **Subtraction or comparison between pointers to different arrays**.

## Examples

### Subtraction or comparison between pointers to different arrays

#### Issue

**Subtraction or comparison between pointers to different arrays** occurs when you subtract or compare pointers that are null or that point to elements in different arrays. The relational operators for the comparison are >, <, >=, and <=.

#### Risk

When you subtract two pointers to elements in the same array, the result is the difference between the subscripts of the two array elements. Similarly, when you compare two pointers to array elements, the result is the positions of the pointers relative to each other. If the pointers are null or point to different arrays, a subtraction or comparison operation is undefined. If you use the subtraction result as a buffer index, it can cause a buffer overflow.

#### Fix

Before you subtract or use relational operators to compare pointers to array elements, check that they are non-null and that they point to the same array.

#### Example - Subtraction Between Pointers to Elements in Different Arrays

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int end;
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation is undefined unless array nums
    is adjacent to variable end in memory. */
    free_elements = &end - next_num_ptr;
    return free_elements;
}
```

In this example, the array `nums` is incrementally filled. Pointer subtraction is then used to determine how many free elements remain. Unless `end` points to a memory location one past the last element of `nums`, the subtraction operation is undefined.

**Correction — Subtract Pointers to the Same Array**

Subtract the pointer to the last element that was filled from the pointer to the last element in the array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation involves pointers to the same array. */
    free_elements = &(nums[SIZE20 - 1]) - next_num_ptr;

    return free_elements + 1;
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [resident]

Using identifiers that are reserved for the implementation

## Description

### Rule Definition

*Using identifiers that are reserved for the implementation.*

### Polyspace Implementation

This checker checks for **Declaration of reserved identifiers or macro names**.

## Examples

### Declaration of reserved identifiers or macro names

#### Issue

The issue occurs when a reserved identifier or macro name is declared.

If you define a macro name that corresponds to a standard library macro, object, or function, rule 21.1 is violated.

The rule considers tentative definitions as definitions.

#### Risk

The Standard allows implementations to treat reserved identifiers specially. If you reuse reserved identifiers, you can cause undefined behavior.

## Check Information

**Decidability:** Decidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [restrict]

Passing pointers into the same object as arguments to different restrict-qualified parameters

## Description

### Rule Definition

*Passing pointers into the same object as arguments to different restrict-qualified parameters.*

### Polyspace Implementation

This checker checks for **Copy of overlapping memory**.

## Examples

### Copy of overlapping memory

#### Issue

**Copy of overlapping memory** occurs when there is a memory overlap between the source and destination argument of a copy function such as `memcpy` or `strcpy`. For instance, the source and destination arguments of `strcpy` are pointers to different elements in the same string.

#### Risk

If there is memory overlap between the source and destination arguments of copy functions, according to C standards, the behavior is undefined.

#### Fix

Determine if the memory overlap is what you want. If so, find an alternative function. For instance:

- If you are using `memcpy` to copy values from one memory location to another, use `memmove` instead of `memcpy`.
- If you are using `strcpy` to copy one string to another, use `memmove` instead of `strcpy`, as follows:

  ```
  s = strlen(source);
  memmove(destination, source, s + 1);
  ```

  `strlen` determines the string length without the null terminator. Therefore, you must move `s+1` bytes instead of `s` bytes.

#### Example - Overlapping Copy

```
#include <string.h>

char str[] = {"ABCDEFGH"};

void my_copy() {
    strcpy(&str[0],(const char*)&str[2]);
}
```

In this example, because the source and destination argument are pointers to the same string `str`, there is memory overlap between their allowed buffers.

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [sigcall]

Calling signal from interruptible signal handlers

## Description

### Rule Definition

*Calling signal from interruptible signal handlers.*

### Polyspace Implementation

This checker checks for **Signal call from within signal handler**.

## Examples

### Signal call from within signal handler

#### Issue

**Signal call from within signal handler** occurs when you call `signal()` from a nonpersistent signal handler on a Windows platform.

#### Risk

A nonpersistent signal handler is reset after catching a signal. The handler does not catch subsequent signals unless the handler is reestablished by calling `signal()`. A nonpersistent signal handler on a Windows platform is reset to `SIG_DFL`. If another signal interrupts the execution of the handler, that signal can cause a race condition between `SIG_DFL` and the existing signal handler. A call to `signal()` can also result in an infinite loop inside the handler.

#### Fix

Do not call `signal()` from a signal handler on Windows platforms.

#### Example - `signal()` Called from Signal Handler

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;

void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;

    /* Call signal() to reestablish sig_handler
    upon receiving SIG_ERR. */

    if (signal(s0, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
}

void func(void)
{
        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
  /* more code */
}
```

In this example, the definition of `sig_handler()` includes a call to `signal()` when the handler catches `SIG_ERR`. On Windows platforms, signal handlers are nonpersistent. This code can result in a race condition.

#### Correction — Do Not Call `signal()` from Signal Handler

If your code requires the use of a persistent signal handler on a Windows platform, use a persistent signal handler after performing a thorough risk analysis.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;


void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;
    /* No call to signal() */
}

int main(void)
{

        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
}
```

## Check Information

**Decidability:** Undecidable

## Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [signconv]

Conversion of signed characters to wider integer types before a check for EOF

## Description

### Rule Definition

*Conversion of signed characters to wider integer types before a check for EOF.*

### Polyspace Implementation

This checker checks for **Misuse of sign-extended character value**.

## Examples

### Misuse of sign-extended character value

#### Issue

**Misuse of sign-extended character value** occurs when you convert a signed or plain `char` data type to a wider integer data type with sign extension. You then use the resulting sign-extended value as array index, for comparison with EOF or as argument to a character-handling function.

#### Risk

*Comparison with EOF*: Suppose, your compiler implements the plain `char` type as signed. In this implementation, the character with the decimal form of 255 (–1 in two's complement form) is stored as a signed value. When you convert a `char` variable to the wider data type `int` for instance, the sign bit is preserved (sign extension). This sign extension results in the character with the decimal form 255 being converted to the integer –1, which cannot be distinguished from EOF.

*Use as array index*: By similar reasoning, you cannot use sign-extended plain `char` variables as array index. If the sign bit is preserved, the conversion from `char` to `int` can result in negative integers. You must use positive integer values for array index.

*Argument to character-handling function*: By similar reasoning, you cannot use sign-extended plain `char` variables as arguments to character-handling functions declared in `ctype.h`, for instance, `isalpha()` or `isdigit()`. According to the C11 standard (Section 7.4), if you supply an integer argument that cannot be represented as `unsigned char` or EOF, the resulting behavior is undefined.

#### Fix

Before conversion to a wider integer data type, cast the signed or plain `char` value explicitly to `unsigned char`.

#### Example - Sign-Extended Character Value Compared with EOF

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];
```

```
static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = *buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

In this example, the function `parser` can traverse a string input `buf`. If a character in the string has the decimal form 255, when converted to the `int` variable `c`, its value becomes –1, which is indistinguishable from `EOF`. The later comparison with `EOF` can lead to a false positive.

#### Correction — Cast to `unsigned char` Before Conversion

One possible correction is to cast the plain `char` value to `unsigned char` before conversion to the wider `int` type.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = (unsigned char)*buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [sizeofptr]

Taking the size of a pointer to determine the size of the pointed-to type

## Description

### Rule Definition

*Taking the size of a pointer to determine the size of the pointed-to type.*

### Polyspace Implementation

This checker checks for **Possible misuse of sizeof**.

## Examples

### Possible misuse of sizeof

#### Issue

**Possible misuse of sizeof** occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

#### Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

#### Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

#### Example - `sizeof` Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

#### Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

## Check Information

**Decidability:** Decidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

#### Topics

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [strmod]

Modifying string literals

## Description

### Rule Definition

*Modifying string literals.*

### Polyspace Implementation

This checker checks for **Writing to const qualified object**.

## Examples

### Writing to const qualified object

#### Issue

**Writing to `const` qualified object** occurs when you do one of the following:

- Use a `const`-qualified object as the destination of an assignment.
- Pass a `const`-qualified object to a function that modifies the argument.

For instance, the defect can occur in the following situations:

- You pass a `const`-qualified object as first argument of one of the following functions:
  - `mkstemp`
  - `mkostemp`
  - `mkostemps`
  - `mkdtemp`
- You pass a `const`-qualified object as the destination argument of one of the following functions:
  - `strcpy`
  - `strncpy`
  - `strcat`
  - `memset`
- You perform a write operation on a `const`-qualified object.

#### Risk

The risk depends upon the modifications made to the `const`-qualified object.

| Situation | Risk |
|---|---|
| Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on. | These functions replace the last six characters of their first argument with a string. Therefore, they expect a modifiable `char` array as their first argument. |
| Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on. | These functions modify their destination argument. Therefore, they expect a modifiable `char` array as their destination argument. |
| Writing to the object | The `const` qualifier implies an agreement that the value of the object will not be modified. By writing to a `const`-qualified object, you break the agreement. The result of the operation is undefined. |

#### Fix

The fix depends on the modification made to the `const`-qualified object.

| Situation | Fix |
|---|---|
| Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on. | Pass a non-`const` object as first argument of the function. |
| Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on. | Pass a non-`const` object as destination argument of the function. |
| Writing to the object | Perform the write operation on a non-`const` object. |

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Writing to `const`-Qualified Object

```
#include <string.h>

const char* buffer = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

In this example, because `buffer` is `const`-qualified, `strchr(buffer,'X')` returns a `const`-qualified `char*` pointer. When this `char*` pointer is used as the destination argument of `strcpy`, a **Writing to const qualified object** error appears.

#### Correction — Copy const-Qualified Object to Non-const Object

One possible correction is to assign the constant string to a non-const object and use the non-const object as destination argument of strchr.

```
#include <string.h>

char buffer[] = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

Check ISO/IEC TS 17961 (-iso-17961)

### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [swtchdflt]

Use of an implied default in a switch statement

## Description

### Rule Definition

*Use of an implied default in a switch statement.*

### Polyspace Implementation

This checker checks for **Missing case for switch condition**.

## Examples

### Missing case for switch condition

#### Issue

**Missing case for switch condition** occurs when the `switch` variable can take values that are not covered by a `case` statement.

**Note** Bug Finder only raises a defect if the switch variable is not full range.

#### Risk

If the `switch` variable takes a value that is not covered by a `case` statement, your program can have unintended behavior.

A switch-statement that makes a security decision is particularly vulnerable when all possible values are not explicitly handled. An attacker can use this situation to deviate the normal execution flow.

#### Fix

It is good practice to use a `default` statement as a catch-all for values that are not covered by a `case` statement. Even if the `switch` variable takes an unintended value, the resulting behavior can be anticipated.

#### Example - Missing Default Condition

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}

int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    }

    printf("Welcome!\n");
    return r;
}
```

In this example, the `enum` parameter `User` can take a value `UNKNOWN` that is not covered by a `case` statement.

**Correction — Add a Default Condition**

One possible correction is to add a default condition for possible values that are not covered by a `case` statement.

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}
```

```
int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    break;
    default:
        printf("Invalid login credentials!\n");
    }

    printf("Welcome!\n");
    return r;
}
```

### Check Information

**Decidability:** Decidable

## Version History

**Introduced in R2019a**

### See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

#### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [syscall]

Calling system

## Description

### Rule Definition

*Calling system.*

### Polyspace Implementation

This checker checks for **Unsafe call to a system function**.

## Examples

### Unsafe call to a system function

#### Issue

**Unsafe call to a system function** occurs when you use a function that invokes an implementation-defined command processor. These functions include:

- The C standard `system()` function.
- The POSIX `popen()` function.
- The Windows `_popen()` and `_wpopen()` functions.

#### Risk

If the argument of a function that invokes a command processor is not sanitized, it can cause exploitable vulnerabilities. An attacker can execute arbitrary commands or read and modify data anywhere on the system.

#### Fix

Do not use a `system`-family function to invoke a command processor. Instead, use safer functions such as POSIX `execve()` and WinAPI `CreateProcess()`.

#### Example - `system()` Called

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};

void func(char *arg)
{
    char buf[SIZE512];
    int retval=sprintf(buf, "/usr/bin/any_cmd %s", arg);
    if (retval<=0 || retval>SIZE512){
        /* Handle error */
        abort();
    }
    /* Use of system() to pass any_cmd with
    unsanitized argument to command processor */

    if (system(buf) == -1) {
    /* Handle error */
  }
}
```

In this example, `system()` passes its argument to the host environment for the command processor to execute. This code is vulnerable to an attack by command-injection.

#### Correction — Sanitize Argument and Use execve()

In the following code, the argument of `any_cmd` is sanitized, and then passed to `execve()` for execution. `exec`-family functions are not vulnerable to command-injection attacks.

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};


void func(char *arg)
{
  char *const args[SIZE3] = {"any_cmd", arg, NULL};
  char  *const env[] = {NULL};

  /* Sanitize argument */

  /* Use execve() to execute any_cmd. */

  if (execve("/usr/bin/time", args, env) == -1) {
    /* Handle error */
  }
}
```

### Check Information

**Decidability:** Undecidable

## Version History

**Introduced in R2019a**

### See Also

Check ISO/IEC TS 17961 (-iso-17961)

**Topics**
“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [taintformatio]

Using a tainted value to write to an object using a formatted input or output function

## Description

### Rule Definition

*Using a tainted value to write to an object using a formatted input or output function.*

### Polyspace Implementation

This checker checks for these issues:

- **Buffer overflow from incorrect string format specifier**.
- **Destination buffer overflow in string manipulation**.
- **Invalid use of standard library routine**.
- **Invalid use of standard library string routine**.
- **Tainted NULL or non-null-terminated string**.
- **Tainted string format specifier**.
- **Invalid use of standard library string routine**.
- **Use of dangerous standard function**.
- **Insufficient destination buffer size**

### Extend Checker

A default Bug Finder analysis might not flag a **Tainted NULL or non-null-terminated string** or **Tainted string format specifier** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Buffer overflow from incorrect string format specifier

#### Issue

**Buffer overflow from incorrect string format specifier** occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

#### Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

#### Fix

Use a format specifier that is compatible with the memory buffer size.

#### Example - Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

#### Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

### Destination buffer overflow in string manipulation

#### Issue

**Destination buffer overflow in string manipulation** occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

#### Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Invalid use of standard library routine

#### Issue

This issue occurs when you use invalid arguments with a function from the standard library. This defect picks up errors related to other functions not covered by float, integer, memory, or string standard library routines.

#### Risk

Invalid arguments to a standard library function result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. For instance, the argument to a `printf` function can be `NULL` because a pointer was initialized with `NULL` and the initialization value was not overwritten along a specific execution path.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example – Calling `printf` Without a String

```
#include <stdio.h>
#include <stdlib.h>

void print_null(void) {

  printf(NULL);
}
```

The function `printf` takes only string input arguments or format specifiers. In this function, the input value is NULL, which is not a valid string.

#### Correction — Use Compatible Input Arguments

One possible correction is to change the input arguments to fit the requirements of the standard library routine. In this example, the input argument was changed to a character.

```
#include <stdio.h>

void print_null(void) {
    char zero_val = '0';
    printf((const char*)zero_val);
}
```

### Invalid use of standard library string routine

#### Issue

**Invalid use of standard library string routine** occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source);
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";
  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

#### Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Tainted NULL or non-null-terminated string

#### Issue

This issue occurs when strings from nonsecure sources are used in string manipulation routines that implicitly dereference the string buffer, for instance, `strcpy` or `sprintf`.

The checker raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

#### Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

#### Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

**Example – Getting String from Input Argument**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}



void errorMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Error: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

**Correction 1 — Validate the Data**

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // Defect only raised here
        // - string is not null
        // - string has a positive and limited size
        // - TAINTED_STRING on strlen used as a firewall
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

**Correction 2 — Validate the Data**

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

### Tainted string format specifier

**Issue**

This issue occurs when `printf`-style functions use a format specifier constructed from nonsecure sources.

**Risk**

If you use externally controlled elements to format a string, you can cause buffer overflow or data-representation problems. An attacker can use these string formatting elements to view the contents of a stack using `%x` or write to a stack using `%n`.

#### Fix

Pass a static string to format string functions. This fix ensures that an external actor cannot control the string.

Another possible fix is to allow only the expected number of arguments. If possible, use functions that do not support the vulnerable `%n` operator in format strings.

#### Example – Get Elements from User Input

```
#include <stdio.h>
#include <unistd.h>
#define MAX 40
void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf(userstr);
}
```

This example prints the input argument `userstr`. The string is unknown. If it contains elements such as `%`, `printf` can interpret `userstr` as a string format instead of a string, causing your program to crash.

#### Correction — Print as String

One possible correction is to print `userstr` explicitly as a string so that there is no ambiguity.

```
#include <stdio.h>
#include <unistd.h>
#define MAX 40
void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf("%.20s", userstr);;
}
```

### Use of dangerous standard function

#### Issue

The **Use of dangerous standard function** check highlights uses of functions that are inherently dangerous or potentially dangerous given certain circumstances. The following table lists possibly dangerous functions, the risks of using each function, and what function to use instead.

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `gets` | Inherently dangerous — You cannot control the length of input from the console. | `fgets` |
| `cin` | Inherently dangerous — You cannot control the length of input from the console. | Avoid or prefaces calls to `cin` with `cin.width`. |
| `strcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `strncpy` |

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `stpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `stpncpy` |
| `lstrcpy` or `StrCpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy` |
| `strcat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `strncat`, `strlcat`, or `strcat_s` |
| `lstrcat` or `StrCat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat` |
| `wcpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcpncpy` |
| `wcscat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `wcsncat`, `wcslcat`, or `wcncat_s` |
| `wcscpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcsncpy` |
| `sprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `snprintf` |
| `vsprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `vsnprintf` |

#### Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.

- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Using `sprintf`

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

#### Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

### Insufficient destination buffer size

#### Issue

**Insufficient destination buffer size** occurs when the destination buffer in a `strcpy` operation cannot accommodate the source buffer and a null terminator. This issue is reported if the size of the source buffer is unknown. Consider this code:

```
int main (int argc, char *argv[])
{
  const char *const input = ((argc && argv[0]) ? argv[0] : "");
  char str[100];
  strcpy(str, input); // Noncompliant
}
```

In this case, the size of the source buffer `input` is unknown. The size of the destination buffer `str` might be smaller than the value `(strlen(input)+1)`. Polyspace reports a violation on the `strcpy` operation.

#### Risk

Using a destination buffer of insufficient size might allow an attacker to cause a buffer overflow. In the preceding code example, if `argv[0]` contains 100 or more characters, the `strcpy` operation results in a buffer overflow.

#### Fix

Before calling the function `strcpy()`, allocate sufficient memory dynamically. For instance, use the function `strlen()` to determine the size of the source buffer and then allocate the destination buffer so that its size is greater than the value `strlen(source) + 1`.

Alternatively, use the function `strncpy()` instead of `strcpy()`. The function `strncpy()` copies a known number of characters from the source buffer to the destination buffer. Because the function `strncpy()` copies a known number of characters, it is a safer alternative to `strcpy()`.

#### Example — Destination Buffer Too Small

In this example, the size of the `source` buffer is unknown, while the size of the `destination` buffer is fixed at `128`. The size of the `destination` buffer might not be sufficient to accommodate the characters from the `source` buffer and terminate the buffer with a null. Polyspace reports a violation of the rule.

```
#include <string.h>

int main(int argc, char *argv[]) {
  const char *const source = (argc && argv[0]) ? argv[0] : "";
  char destination[128];
  strcpy(destination,source);//Noncompliant

  return 0;
}
```

#### Correction — Use `strncpy` instead of `strcpy`

To fix this issue, use `strncpy()` to copy a known number of characters from the `source` buffer to the `destination` buffer.

```
#include <string.h>
#define MAX_ARGS  128
```

```
int main(int argc, char *argv[]) {
  const char *const source = (argc && argv[0]) ? argv[0] : "";
  char destination[MAX_ARGS];
  int num = (strlen(source)+1>MAX_ARGS)?MAX_ARGS:strlen(source)+1;
  strncpy(destination,source,num);//Compliant

  return 0;
}
```

**Correction — Allocate Sufficient Memory for Destination Buffer**

This violation is resolved by allocating sufficient memory for the `destination` buffer. For instance, use the function `strlen()` to calculate the size of the `source` buffer and allocate sufficient memory for the `destination` buffer so that it can accommodate all characters from the `source` buffer and the null terminator (`'\0'` ).

```
#include <string.h>

int main(int argc, char *argv[]) {
    const char *const source = (argc && argv[0]) ? argv[0] : "";
    char* destination = (char *)malloc(strlen(source)+ 1);
    if(destination!=NULL){
        strcpy(destination, source);//Compliant
    }else{
        /*Handle Error*/
    }
    //...
    free(destination);
    return 0;
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [taintnoproto]

Using a tainted value as an argument to an unprototyped function pointer

## Description

### Rule Definition

*Using a tainted value as an argument to an unprototyped function pointer.*

### Polyspace Implementation

This checker checks for **Call through non-prototyped function pointer**.

## Examples

### Call through non-prototyped function pointer

#### Issue

**Call through non-prototyped function pointer** detects a call to a function through a pointer without a prototype. A function prototype specifies the type and number of parameters.

#### Risk

Arguments passed to a function without a prototype might not match the number and type of parameters of the function definition, which can cause undefined behavior. If the parameters are restricted to a subset of their type domain, arguments from untrusted sources can trigger vulnerabilities in the called function.

#### Fix

Before calling the function through a pointer, provide a function prototype.

#### Example - Argument Does Not Match Parameter Restriction

```
#include <stdio.h>
#include <limits.h>
#define SIZE2 2

typedef void (*func_ptr)();
extern int getchar_wrapper(void);
extern void restricted_int_sink(int i);
/* Integer value restricted to
range [-1, 255] */
extern void restricted_float_sink(double i);
/* Double value restricted to > 0.0 */



func_ptr generic_callback[SIZE2] =
{
    (func_ptr)restricted_int_sink,
    (func_ptr)restricted_float_sink
};
```

```
void func(void)
{
    int ic;
    ic = getchar_wrapper();
    /* Wrong index used for generic_callback.
    Negative 'int' passed to restricted_float_sink. */
    (*generic_callback[1])(ic);
}
```

In this example, a call through `func_ptr` passes `ic` as an argument to function `generic_callback[1]`. The type of `ic` can have negative values, while the parameter of `generic_callback[1]` is restricted to float values greater than `0.0`. Typically, compilers and static analysis tools cannot perform type checking when you do not provide a pointer prototype.

#### Correction — Provide Prototype of Pointer to Function

Pass the argument `ic` to a function with a parameter of type `int`, by using a properly prototyped pointer.

```
#include <stdio.h>
#include <limits.h>
#define SIZE2 2

typedef void (*func_ptr_proto)(int);
extern int getchar_wrapper(void);
extern void restricted_int_sink(int i);
/* Integer value restricted to
range [-1, 255] */
extern void restricted_float_sink(double i);
/* Double value restricted to > 0.0 */

func_ptr_proto generic_callback[SIZE2] =
{
    (func_ptr_proto)restricted_int_sink,
    (func_ptr_proto)restricted_float_sink
};

void func(void)
{
    int ic;
    ic = getchar_wrapper();
    /* ic passed to function through
properly prototyped pointer. */
    (*generic_callback[0])(ic);
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [taintsink]

Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink

## Description

### Rule Definition

*Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink.*

### Polyspace Implementation

This checker checks for these issues:

- **Tainted size of variable length array**.
- **Pointer dereference with tainted offset**.
- **Array access with tainted index**.

### Extend Checker

A default Bug Finder analysis might not flag a **Tainted size of variable length array**, **Pointer dereference with tainted offset**, or **Array access with tainted index** issue for certain inputs that originate outside of the current analysis boundary. See "Sources of Tainting in a Polyspace Analysis". To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Tainted size of variable length array

#### Issue

**Tainted size of variable length array** detects variable length arrays (VLA) whose size is from an unsecure source.

#### Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

#### Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

#### Example - Input Argument Used as Size of VLA

```
#include<stdio.h>
#inclule<stdlib.h>
```

```
#define LIM 40

long squaredSum(int size) {

    int tabvla[size];
    long res = 0;
    for (int i=0 ; i<LIM-1 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

#### Correction — Check VLA Size

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 0 and less than 40, before creating the VLA

```
#include <stdio.h>
#include <stdlib.h>
#define LIM 40

long squaredSum(int size) {
    long res = 0;
    if (size>0 && size<LIM){
        int tabvla[size];
        for (int i=0 ; i<size || i<LIM-1 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }else{
        res = -1;
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

### Pointer dereference with tainted offset

#### Issue

**Pointer dereference with tainted offset** detects pointer dereferencing, either reading or writing, using an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

#### Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

#### Example - Dereference Pointer Array

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset];
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of the index. If the index is inside the valid range, continue with the pointer dereferencing.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

### Array access with tainted index

#### Issue

**Array access with tainted index** detects reading or writing to an array by using a tainted index that has not been validated.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

#### Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

#### Example - Use Index to Return Buffer Value

```
#include <stdlib.h>
#include <stdio.h>
```

```
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];
}
```

In this example, the index `num` accesses the array `tab`. The function does not check to see if `num` is inside the range of `tab`.

**Correction — Check Range Before Use**

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

**Topics**

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [taintstrcpy]

Tainted strings are passed to a string copying function

## Description

### Rule Definition

*Tainted strings are passed to a string copying function.*

### Polyspace Implementation

This checker checks for **Tainted NULL or non-null-terminated string**.

### Extend Checker

A default Bug Finder analysis might not flag a **Tainted NULL or non-null-terminated string** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Tainted NULL or non-null-terminated string

#### Issue

**Tainted NULL or non-null-terminated string** looks for strings from unsecure sources that are being used in string manipulation routines that implicitly dereference the string buffer. For example, `strcpy` or `sprintf`.

**Tainted NULL or non-null-terminated string** raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

**Note** If you reference a string using the form `ptr[i]`, `*ptr`, or pointer arithmetic, Bug Finder raises a **Use of tainted pointer** defect instead. The **Tainted NULL or non-null-terminated string** defect is raised only when the pointer is used as a string.

#### Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

#### Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

**Example - Getting String from Input Argument**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

**Correction — Validate the Data**

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // Defect only flagged here
        // - string is not null
        // - string has a positive and limited size
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

#### Correction — Validate the Data

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

#### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [uninitref]

Referencing uninitialized memory

## Description

### Rule Definition

*Referencing uninitialized memory.*

### Polyspace Implementation

This checker checks for these issues:

- **Non-initialized pointer**.
- **Pointer to non-initialized value converted to const pointer**.
- **Non-initialized variable**.

### Extend Checker

You can extend the checker in the following ways:

- Polyspace does not flag passing pointers to noninitialized variables to functions. To detect noninitialized variables that are passed to functions by pointers, extend the checker by using the option `-code-behavior-specification`. See "Extend Checkers for Initialization to Check Function Arguments Passed by Pointers".
- If a variable in your code is non-initialized only for certain system input values, you can see one possible combination of input values causing the defect. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

## Examples

### Non-initialized pointer

#### Issue

**Non-initialized pointer** occurs when a pointer is not assigned an address before dereference.

#### Risk

Unless a pointer is explicitly assigned an address, it points to an unpredictable location.

#### Fix

The fix depends on the root cause of the defect. For instance, you assigned an address to the pointer but the assignment is unreachable.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface".

See examples of fixes below. It is a good practice to initialize a pointer to NULL when declaring the pointer.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Example - Non-initialized pointer error

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL) 
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j;                    
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

### Correction — Initialize Pointer on Every Execution Path

One possible correction is to assign an address to `pi` when `prev` is not `NULL`.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL) 
       {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
       } 
    /* Fix: Initialize pi in branches of if statement  */
    else 
        pi = prev;              


    *pi = j;
    return pi;
}
```

### Pointer to non-initialized value converted to const pointer

#### Issue

**Pointer to non initialized value converted to const pointer** occurs when a pointer to a constant (`const int*`, `const char*`, etc.) is assigned an address that does not yet contain a value.

#### Risk

A pointer to a constant stores a value that must not be changed later in the program. If you assign the address of a non-initialized variable to the pointer, it now points to an address with garbage values for the remainder of the program.

#### Fix

Initialize a variable before assigning its address to a pointer to a constant.

#### Example - Pointer to non initialized value converted to const pointer error

```
#include<stdio.h>

void Display_Parity()
 {
  int num,parity;
  const int* num_ptr = &num;
  /* Defect: Address &num does not store a value */

  printf("Enter a number\n:");
  scanf("%d",&num);

  parity=((*num_ptr)%2);
  if(parity==0)
    printf("The number is even.");
  else
    printf("The number is odd.");

 }
```

`num_ptr` is declared as a pointer to a constant. However the variable `num` does not contain a value when `num_ptr` is assigned the address `&num`.

#### Correction — Store Value in Address Before Assignment to Pointer

One possible correction is to obtain the value of `num` from the user before `&num` is assigned to `num_ptr`.

```
#include<stdio.h>

void Display_Parity()
 {
  int num,parity;
  const int* num_ptr;

  printf("Enter a number\n:");
  scanf("%d",&num);

 /* Fix: Assign &num to pointer after it receives a value */
  num_ptr=&num;
  parity=((*num_ptr)%2);
  if(parity==0)
    printf("The number is even.");
  else
    printf("The number is odd.");
 }
```

The `scanf` statement stores a value in `&num`. Once the value is stored, it is legitimate to assign `&num` to `num_ptr`.

### Non-initialized variable

#### Issue

**Non-initialized variable** occurs when a variable is not initialized before its value is read.

#### Risk

Unless a variable is explicitly initialized, the variable value is unpredictable. You cannot rely on the variable having a specific value.

#### Fix

The fix depends on the root cause of the defect. For instance, you assigned a value to the variable but the assignment is unreachable or you assigned a value to the variable in one of two branches of a conditional statement. Fix the unreachable code or missing assignment.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below. It is a good practice to initialize a variable at declaration.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
```

```
      {
        val = getsensor();
      }

    return val;
    /* Defect: val does not have a value if command is not 2 */
}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

#### Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that the initialization is not bypassed on some execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

### Topics

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [usrfmt]

Including tainted or out-of-domain input in a format string

## Description

### Rule Definition

*Including tainted or out-of-domain input in a format string.*

### Polyspace Implementation

This checker checks for these issues:

- **Format string specifiers and arguments mismatch**
- **Tainted string format**

.

### Extend Checker

A default Bug Finder analysis might not flag a **Tainted string format** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Format string specifiers and arguments mismatch

#### Issue

**Format string specifiers and arguments mismatch** occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

#### Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

#### Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
  const char *message = "License not available";
  int err_number = ;-4
  printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

#### Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

#### Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

### Tainted string format

#### Issue

**Tainted string format** occurs when `printf`-style functions use a format specifier constructed from unsecure sources.

#### Risk

If you use externally controlled elements to format a string, you can cause buffer overflow or data-representation problems. An attacker can use these string formatting elements to view the contents of a stack using `%x` or write to a stack using `%n`.

#### Fix

Pass a static string to format string functions. This fix ensures that an external actor cannot control the string.

Another possible fix is to allow only the expected number of arguments. If possible, use functions that do not support the vulnerable `%n` operator in format strings

#### Example — Get Elements from User Input

```
#include <stdio.h>
#include <unistd.h>
#define MAX 40
void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf(userstr);//Noncompliant
}
```

This example prints the input argument `userstr`. The string is unknown. If it contains elements such as `%`, `printf` can interpret `userstr` as a string format instead of a string, causing your program to crash.

#### Correction — Print as String

One possible correction is to print `userstr` explicitly as a string so that there is no ambiguity.

```
#include "stdio.h"
#include <unistd.h>
#define MAX 40

void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf("%.20s", userstr);
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

#### Topics

"Check for and Review Coding Standard Violations"

# ISO/IEC TS 17961 [xfilepos]

Using a value for fsetpos other than a value returned from fgetpos

## Description

### Rule Definition

*Using a value for fsetpos other than a value returned from fgetpos.*

### Polyspace Implementation

This checker checks for **Invalid file position**.

## Examples

### Invalid file position

#### Issue

**Invalid file position** occurs when the file position argument of `fsetpos()` uses a value that is not obtained from `fgetpos()`.

#### Risk

The function `fgetpos(FILE *stream, fpos_t *pos)` gets the current file position of the stream. When you use any other value as the file position argument of `fsetpos(FILE *stream, const fpos_t *pos)`, you might access an unintended location in the stream.

#### Fix

Use the value returned from a successful call to `fgetpos()` as the file position argument of `fsetpos()`.

#### Example - `memset()` Sets File Position Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset' */
    (void)memset(&offset, 0, sizeof(offset));

    /* Read data from file */

    /* Return to the initial position. offset was not
    returned from a call to fgetpos()    */
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

In this example, `fsetpos()` uses `offset` as its file position argument. However, the value of `offset` is set by `memset()`. The preceding code might access the wrong location in the stream.

#### Correction — Use a File Position Returned From `fgetpos()`

Call `fgetpos()`, and if it returns successfully, use the position argument in your call to `fsetpos()`.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset'
    using fgetpos() */
    if (fgetpos(file, &offset) != 0)
    {
        /* Handle error */
    }

    /* Read data from file */

    /* Back to the initial position */
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

`Check ISO/IEC TS 17961 (-iso-17961)`

**Topics**

“Check for and Review Coding Standard Violations”

# ISO/IEC TS 17961 [xfree]

Reallocating or freeing memory that was not dynamically allocated

## Description

### Rule Definition

*Reallocating or freeing memory that was not dynamically allocated.*

### Polyspace Implementation

This checker checks for **Invalid free of pointer**.

## Examples

### Invalid free of pointer

#### Issue

**Invalid free of pointer** occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

#### Risk

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function or a previous `malloc` function on a different pointer.

#### Fix

In most cases, you can fix the issue by removing the `free` statement. If the pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

#### Example - Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer p is deallocated using the free function. However, p points to a memory location that was not dynamically allocated.

**Correction — Remove Pointer Deallocation**

If the number of elements of the array p is known at compile time, one possible correction is to remove the deallocation of the pointer p.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

**Correction — Introduce Pointer Allocation**

If the number of elements of the array p is not known at compile time, one possible correction is to dynamically allocate memory to the array p.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

## Check Information

**Decidability:** Undecidable

# Version History

**Introduced in R2019a**

## See Also

Check ISO/IEC TS 17961 (-iso-17961)

### Topics

"Check for and Review Coding Standard Violations"

24

# MISRA C++: 2008

# MISRA C++:2008 Rule 0-1-1

A project shall not contain unreachable code

## Description

### Rule Definition

*A project shall not contain unreachable code.*

### Rationale

This rule flags situations where a group of statements is unreachable because of syntactic reasons. For instance, code following a `return` statement are always unreachable.

Unreachable code involve unnecessary maintenance and can often indicate programming errors.

### Polyspace Implementation

Polyspace reports a defect if a statement in your code is not reachable.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Unreachable statements

```
int func(int arg) {
 int temp = 0;
 switch(arg) {
     temp = arg; // Noncompliant
     case 1:
     {
         break;
     }
     default:
     {
         break;
     }
 }
 return arg;
 arg++; // Noncompliant
}
```

These statements are unreachable:

- Statements inside a `switch` statement that do not belong to a `case` or `default` block.
- Statements after a `return` statement.

## Check Information

**Group:** Language Independent Issues
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 0-1-2

A project shall not contain infeasible paths

## Description

### Rule Definition

*A project shall not contain infeasible paths.*

### Rationale

This rule flags situations where a group of statements is redundant because of nonsyntactic reasons. For instance, an `if` condition is always true or false. Code that is unreachable from syntactic reasons are flagged by rule 0-1-1.

Unreachable or redundant code involve unnecessary maintenance and can often indicate programming errors.

### Polyspace Implementation

Polyspace Bug Finder reports a violation of this if your code contains dead code and unnecessary `if` conditionals. See `Dead code` and `Useless if` checkers.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Boolean Operations with Invariant Results

```
void func (unsigned int arg) {
 if (arg >= 0U) //Noncompliant
     arg  = 1U;
 if (arg < 0U) //Noncompliant
     arg = 1U;
}
```

An `unsigned int` variable is nonnegative. Both `if` conditions involving the variable are always true or always false and are therefore redundant.

## Check Information

**Group:** Language Independent Issues
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 0-1-3

A project shall not contain unused variables

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis . See “Checkers Deactivated in Polyspace as You Code Analysis”.*

### Rule Definition

*A project shall not contain unused variables.*

### Rationale

Presence of unused variables indicates that the wrong variable name might be used in the source code. Removing these variables reduces the possibility of the wrong variable being used in further development. Keep padding bits in bitfields unnamed to reduce unused variables in your project.

### Polyspace Implementation

The checker flags local or global variables that are declared or defined but not read or written in any source files of the project. This specification also applies to members of structures and classes.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Named Bit Field for Padding

```
#include <iostream>
struct S {
    unsigned char b1 : 3;
    unsigned char pad: 1;  //Noncompliant
    unsigned char b2 : 4;
};
void init(struct S S_obj)
{
    S_obj.b1 = 0;
    S_obj.b2 = 0;
}
```

In this example, the bit field `pad` is used for padding the structure. Therefore, the field is never read or written and causes a violation of this rule. To avoid the violation, use an unnamed field for padding.

```
#include <iostream>
struct S {
    unsigned char b1 : 3;
    unsigned char : 1;  //Compliant
    unsigned char b2 : 4;
};
void init(struct S S_obj)
{
    S_obj.b1 = 0;
    S_obj.b2 = 0;
}
```

## Check Information

**Group:** Language Independent Issues
**Category:** Required

# Version History

**Introduced in R2018a**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 0-1-4

A project shall not contain non-volatile POD variables having only one use

## Description

### Rule Definition

*A project shall not contain non-volatile POD variables having only one use.*

### Rationale

If you use a non-volatile variable with a Plain Old Data type (`int`, `double`, etc.) *only once*, you can replace the variable with a constant literal. Your use of a variable indicates that you intended more than one use for that variable and might have a programming error in the code. You might have omitted the other uses of the non-volatile variable or incorrectly used other variables at intended points of use.

### Polyspace Implementation

The checker flags local and static variables that have a function scope (locally static) and file scope, which are used only once. The checker considers `const`-qualified global variables without the `extern` specifier as static variables with file scope.

The checker counts these use cases as one use of the non-volatile variable:

- An explicit initialization using a constant literal or the return value of a function
- An assignment
- A reference to the variable such as a read operation
- An assignment of the variable address to a pointer

If the variable address is assigned to a pointer, the checker assumes that the pointer might be dereferenced later and does not flag the variable.

Some objects are designed to be used only once by their semantics. Polyspace does not flag a single use of these objects:

- `lock_guard`
- `scoped_lock`
- `shared_lock`
- `unique_lock`
- `thread`
- `future`
- `shared_future`

If you use nonstandard objects that provide similar functionality as the objects in the preceding list, Polyspace might flag single uses of the nonstandard objects. Justify their single uses by using comments.

#### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

#### Non-volatile Variable Used Only Once

```
#include <mutex>
int readStatus1();
int readStatus2();
void getReading(int*);
extern std::mutex m;
void foo()
{
    // Initiating lock 'lk'
    std::lock_guard<std::mutex> lk{m};
    int checkEngineStatus1 = readStatus1();
    int checkEngineStatus2 = readStatus2();//Noncompliant
    int sensorData;//Noncompliant
    getReading(&sensorData);
    if(checkEngineStatus1) {
        //Perform some actions if both statuses are valid
    }
    // Release lock when 'lk' is deleted at exit point of scope
}
```

In this example, the variable `checkEngineStatus2` is used only once. The single use of this variable might indicate a programming error. For instance, you might have intended to check both `checkEngineStatus1` and `checkEngineStatus2` in the `if` condition, but omitted the second check. The variable `sensorData` is also used only once when its address is passed to the function `getReading()`. Polyspace flags these single use variables.

The `lock_guard` object `lk` is used a single time. Because the semantics of a `lock_guard` object justifies its single use, Polyspace does not flag it.

## Check Information

**Group:** Language Independent Issues
**Category:** Required

# Version History

**Introduced in R2020b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

#### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 0-1-5

A project shall not contain unused type declarations

## Description

### Rule Definition

*A project shall not contain unused type declarations.*

### Rationale

If you declare a type but do not use it, determining if the type is redundant or left unused by mistake is difficult when you review your code.

For example, if you declare an enumerated data type to store some specialized data, but then use an integer type to store the data, the unused type can indicate a coding error.

### Polyspace Implementation

Polyspace reports a violation of this rule if your code contains unused local types.

As an exception, Polyspace does not report a violation if you define the unused type with `public` access in a template class.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unused `enum` Declaration

In this example, you do not use the enumerated type `switchValue`.Because you declare the type but do not use it, Polyspace reports a violation.

```
enum switchValue {low, medium, high}; //Noncompliant

void operate(int userInput) {
    switch(userInput) {
        case 0: // Turn on low setting
                 break;
        case 1: // Turn on medium setting
                 break;
        case 2: // Turn on high setting
                 break;
        default: // Return error
    }
}
```

Perhaps the intention was to use the type as `switch` input like this.

```
enum switchValue {low, medium, high}; //Compliant

void operate(switchValue userInput) {
    switch(userInput) {
        case low: // Turn on low setting
                   break;
        case medium: // Turn on medium setting
                     break;
        case high: // Turn on high setting
                   break;
        default: // Return error
    }
}
```

#### Public Unused Types in Templates

When developing template classes, you might declare a public `typedef` that is used by the clients of the template, even if the template itself does not use the `typedef`. In such a case, Polyspace does not report a violation if a public `typedef` remains unused in a template. For instance, in this code, Polyspace does not report a violation for the unused type `T*`.

```
template<typename T>
  class myContainerIterator {
    public:
      using pointer = T*;              // Compliant
      typedef T* Pointer;              // Compliant

    //...
  };
```

### Check Information

**Group:** Language Independent Issues
**Category:** Required

## Version History

**Introduced in R2018a**

### See Also

`Check MISRA C++:2008 (-misra-cpp)`

#### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 0-1-6

A project shall not contain instances of non-volatile variables being given values that are never subsequently used.

## Description

### Rule Definition

*A project shall not contain instances of non-volatile variables being given values that are never subsequently used.*

### Rationale

If you assign a value to a variable but do not use the variable value subsequently, the assignment might indicate a programming error. Perhaps you forgot to use the variable later or incorrectly used other variables at the intended points of use.

### Polyspace Implementation

The checker flags value assignments to local and static variables with file scope if the assigned values are not subsequently used. The checker considers `const`-qualified global variables without the `extern` specifier as static variables with file scope.

The checker flags:

- Initializations if the initialized variable is not used.
- Non-initialization assignments if the assigned values are not used.

The checker does not flag the situation where an initialization value is immediately overwritten and therefore ends up unused.

The checker does not flag redundant assignments:

- To variables with class type.
- In the last iteration of a loop, if the assignments in the previous iterations are not redundant.

  For instance, the assignment `prevIter = i` in the last iteration of the loop is redundant but the assignments in the previous iterations are not.

  ```
  void doSomething(int);

  void func() {
    int prevIter=-1, uBound=100;
    for(int i=0; i < uBound; i++) {
          doSomething(prevIter);
          prevIter = i;
    }
  }
  ```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Initialization Value Not Used

```
class largeInteger {
       largeInteger(int d1, int d2, int d3):
            lastFiveDigits(d1), nextFiveDigits(d2), firstFiveDigits(d3){}
       largeInteger& operator=(const largeInteger& other) {
            if(&other !=this) {
              firstFiveDigits = other.firstFiveDigits;
              nextFiveDigits = other.nextFiveDigits;
              lastFiveDigits = other.lastFiveDigits;
            }
            return *this;
       }
       void printIntegerValue();
    private:
        int firstFiveDigits;
        int nextFiveDigits;
        int lastFiveDigits;
};

bool compareValues(largeInteger, largeInteger);

void func() {
    largeInteger largeUnit{10000,0,0}; //Compliant
    largeInteger smallUnit{1,0,0}; //Compliant
    largeInteger tinyUnit{0,1,0}; //Noncompliant
    if(compareValues (largeUnit, smallUnit)) {
        //Perform some action
    }
}
```

In this example, the variable `tinyUnit` is initialized but never used.

## Check Information

**Group:** Language independent issues
**Category:** Required

# Version History

**Introduced in R2023a**

## See Also

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 0-1-7

The value returned by a function having a non- void return type that is not an overloaded operator shall always be used

## Description

### Rule Definition

*The value returned by a function having a non- void return type that is not an overloaded operator shall always be used.*

### Rationale

The unused return value might indicate a coding error or oversight.

Overloaded operators are excluded from this rule because their usage must emulate built-in operators which might not use their return value.

### Polyspace Implementation

The checker flags functions with non-`void` return if the return value is not used or not explicitly cast to a `void` type.

The checker does not flag the functions `memcpy`, `memset`, `memmove`, `strcpy`, `strncpy`, `strcat`, `strncat` because these functions simply return a pointer to their first arguments.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unused Return Value

```
#include <iostream>
#include <new>
#include <algorithm>
#include <cstdint>
#include <vector>

int assignMemory(int * ptr){
     int res = 1;
     ptr = new (std::nothrow) int;
     if(ptr==NULL) {
         res = 0;
     }
     return res;
}
void foo()
{
}

void main() {
    int val;
    int status;
    std::vector<std::int8_t> numVec{10,10,20,20,30,40,50,50,60};

    assignMemory(&val);   //Noncompliant
    status = assignMemory(&val); //Compliant
    (void)assignMemory(&val); //Compliant

    numVec.erase(std::unique(numVec.begin(), numVec.end()), numVec.end());// Noncompliant
}
```

In this example, Polyspace flags calls to functions with unused return value:

- Because `main` does not use the return value of the first call to the function `assignMemory`, Polyspace reports a violation.
- Because the return value of the second call to `assignMemory` is assigned to a local variable, it is compliant with this rule.
- Because the third call to `assignMemory` is cast to `void`, it is compliant with this rule
- Because `main` does not use the object returned by the function `std::vector::erase()`, Polyspace reports a violation.

## Check Information

**Group:** Language Independent Issues
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 0-1-8

All functions with void return type shall have external side effect(s)

## Description

### Rule Definition

*All functions with void return type shall have external side effect(s).*

### Rationale

A function that has a `void` return type does not return anything. When such a function has no external side effects, it does not contribute to the output and consumes time. That these functions have no contribution to the output of the program might be contrary to developer expectation. Such a function might be unnecessary or indicate issues in the program design. Avoid such functions.

The MISRA C++:2008 standard considers these effects as external side effects:

- Reading or writing in resources such as a file or a stream.
- Changing the value of a nonlocal variable.
- Changing the value of a reference type argument.
- Using a `volatile` object.
- Raising an exception.

### Polyspace Implementation

Polyspace flags the definition of a `void` type function if the function has no side effects.

Polyspace considers a function to have side effects if the function performs any of these tasks:

- Calls an impure function other than itself.
- Changes the value or dereferences a reference or pointer type argument.
- Changes the value or deferences of a nonlocal variable.
- Uses a `volatile` object.
- Contains assembly instructions.
- Accesses the `this` pointer of its parent class.
- Raises an exception. Raising exceptions is considered as a side effect even if the function does not exit with an exception.
- Dereferences a local pointer that is assigned an absolute address.
- Accesses a `volatile` object, class member, or `struct` member.
- Sets the value of a class or `struct` member.

This checker does not flag placement `new` and `delete` functions.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid void Functions That Have No Side-Effects

```
#include<cmath>

void init ( int refToInt )          // Noncompliant
{
    refToInt = 0;
}

void transform(double theta) //Noncompliant
{
    theta = sin(theta)/(1-cos(theta));
}
void foo(int val){ //Noncompliant
    if(val){
        foo(val--);
    }
}
```

In this example, Polyspace flags the `void` functions that have no side effects. For instance:

- The function `init()` initializes a local variable. This function returns no value and has no additional side effects. Perhaps the function is intended to initialize a reference and you missed an & in the declaration. Polyspace flags the function.
- The function `transform()` transforms a number to another number by calling the pure functions `sin()` and `cos()`. The function returns nothing and has no side effects. Perhaps the function is intended to return the transformed number and you missed a return statement. Polyspace flags the function.
- The function `foo()` sets up a recursion but has no effect on the output. Perhaps the recursion is set up improperly. Polyspace flags the function.

## Check Information

**Group:** Language Independent Issues
**Category:** Required

# Version History

**Introduced in R2022a**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 0-1-9

There shall be no dead code

## Description

### Rule Definition

*There shall be no dead code.*

### Rationale

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code. For instance, suppose that a variable is never read following a write operation. The write operation is redundant.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

### Polyspace Implementation

Polyspace reports violation of this rule on statements that have no effect on the code, such as writing a value to a variable that is not used subsequently.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Redundant Operations

```
#define ULIM 10000

int func(int arg) {
    int res;
    res = arg*arg + arg;
    if (res > ULIM)
        res = 0; //Noncompliant
    return arg;
}
```

In this example, the operations involving `res` are redundant because the function `func` returns its argument `arg`. All operations involving `res` can be removed without changing the effect of the function.

The checker flags the last write operation on `res` because the variable is never read after that point. The dead code can indicate an unintended coding error. For instance, you intended to return the value of `res` instead of `arg`.

## Check Information

**Group:** Language Independent Issues
**Category:** Required

# Version History

**Introduced in R2016b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 0-1-10

Every defined function shall be called at least once

## Description

### Rule Definition

*Every defined function shall be called at least once.*

### Rationale

If a function with a definition is not called, it might indicate a serious coding error. For instance, the function call is unreachable or a different function is called unintentionally.

### Polyspace Implementation

The checker detects situations where a static function is defined but not called at all in its translation unit.

### Additional Message in Report

Every defined function shall be called at least once. The static function *`funcName`* is not called.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Uncalled Static Function

```
static void func1() {
}

static void func2() { //Noncompliant
}

void func3();

int main() {
    func1();
    return 0;
}
```

The `static` function `func2` is defined but not called.

The function `func3` is not called either, however, it is only declared and not defined. The absence of a call to `func3` does not violate the rule.

## Check Information

**Group:** Language Independent Issues
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 0-1-11

There shall be no unused parameters (named or unnamed) in nonvirtual functions

## Description

### Rule Definition

*There shall be no unused parameters (named or unnamed) in nonvirtual functions.*

### Rationale

Unused parameters often indicate later design changes. You perhaps removed all uses of a specific parameter but forgot to remove the parameter from the parameter list.

Unused parameters constitute an unnecessary overhead. You can also inadvertently call the function with a different number of arguments causing a parameter mismatch.

### Polyspace Implementation

The checker flags a function that has unused named parameters unless the function body is empty.

### Additional Message in Report

Function *funcName* has unused parameters.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unused Parameters

```
typedef int (*callbackFn) (int a, int b);

int callback_1 (int a, int b) { //Compliant
    return a+b;
}

int callback_2 (int a, int b) { //Noncompliant
    return a;
}

int callback_3 (int, int b) { //Compliant - flagged by Polyspace
    return b;
}

int getCallbackNumber();
int getInput();

void main() {
    callbackFn ptrFn;
    int n = getCallbackNumber();
    int x = getInput(), y = getInput();
    switch(n) {
        case 0: ptrFn = &callback_1; break;
        case 1: ptrFn = &callback_2; break;
        default: ptrFn = &callback_3; break;
    }

    (*ptrFn)(x,y);
}
```

In this example, the three functions `callback_1`, `callback_2` and `callback_3` are used as callback functions. One of the three functions is called via a function pointer depending on a value obtained at run time.

- Function `callback_1` uses all its parameters and does not violate the rule.
- Function `callback_2` does not use its parameter `a` and violates this rule.
- Function `callback_3` also does not use its first parameter but it does not violate the rule because the parameter is unnamed. However, Polyspace flags the unused parameter as a rule violation. If you see a violation of this kind, justify the violation with comments. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications”.

## Check Information

**Group:** Language Independent Issues
**Category:** Required

# Version History

**Introduced in R2016b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 0-1-12

There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it

## Description

### Rule Definition

*There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.*

### Rationale

Unused parameters often indicate later design changes. You perhaps removed all uses of a specific parameter but forgot to remove the parameter from the parameter list.

Unused parameters constitute an unnecessary overhead. You can also inadvertently call the function with a different number of arguments causing a parameter mismatch.

### Polyspace Implementation

For each virtual function, the checker looks at all overrides of the function. If an override has a named parameter that is not used, the checker shows a violation on the original virtual function and lists the override as a supporting event.

Note that Polyspace checks for unused parameters in virtual functions within single translation units. For instance, if a base class contains a virtual method with an unused parameter but the derived class implementation of the method uses that parameter, the rule is not violated. However, if the base class and derived class are defined in different files, the checker, which operates file by file, flags a violation of this rule on the base class.

The checker does not flag unused parameters in functions with empty bodies.

### Additional Message in Report

There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.

Function *funcName* has unused parameters.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unused Parameter in Virtual Function

```
class base {
    public:
       virtual void assignVal (int arg1, int arg2) = 0; //Noncompliant
       virtual void assignAnotherVal (int arg1, int arg2) = 0;
};

class derived1: public base {
    public:
       virtual void assignVal (int arg1, int arg2) {
           arg1 = 0;
       }
       virtual void assignAnotherVal (int arg1, int arg2) {
           arg1 = 1;
       }
};

class derived2: public base {
    public:
       virtual void assignVal (int arg1, int arg2) {
           arg1 = 0;
       }
       virtual void assignAnotherVal (int arg1, int arg2) {
           arg2 = 1;
       }
};
```

In this example, the second parameter of the virtual method `assignVal` is not used in any of the derived class implementations of the method.

On the other hand, the implementation of the virtual method `assignAnotherVal` in derived class `derived1` uses the first parameter of the method. The implementation in `derived2` uses the second parameter. Both parameters of `assignAnotherVal` are used and therefore the virtual method does not violate the rule.

## Check Information

**Group:** Language Independent Issues
**Category:** Required

# Version History

**Introduced in R2016b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 0-2-1

An object shall not be assigned to an overlapping object

## Description

### Rule Definition

*An object shall not be assigned to an overlapping object.*

### Rationale

When you assign an object to another object with overlapping memory, the behavior is undefined.

The exceptions are:

- You assign an object to another object with exactly overlapping memory and compatible type.
- You copy one object to another with `memmove`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Assignment of Union Members

```
void func (void) {
    union {
        short i;
        int j;
    } a = {0}, b = {1};

    a.j = a.i;   //Noncompliant
    a = b;       //Compliant
}
```

In this example, the rule is violated when `a.i` is assigned to `a.j` because the two variables have overlapping regions of memory.

## Check Information

**Group:** Language Independent Issues
**Category:** Required

# Version History

**Introduced in R2016b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**
“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 0-3-2

If a function generates error information, then that error information shall be tested

## Description

### Rule Definition

*If a function generates error information, then that error information shall be tested.*

### Rationale

If you do not check the return value of functions that indicate error information through their return values, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

For the `errno`-setting functions, to see if the function call completed without errors, check `errno` for error values. The return values of these `errno`-setting functions do not indicate errors. The return value can be one of the following:

- `void`
- Even if an error occurs, the return value can be the same as the value from a successful call. Such return values are called in-band error indicators. For instance, `strtol` converts a string to a long integer and returns the integer. If the result of conversion overflows, the function returns `LONG_MAX` and sets `errno` to `ERANGE`. However, the function can also return `LONG_MAX` from a successful conversion. Only by checking `errno` can you distinguish between an error and a successful conversion.

For the `errno`-setting functions, you can determine if an error occurred only by checking `errno`.

### Polyspace Implementation

The checker raises a violation when:

- You call sensitive functions that return information about possible errors and then you ignore the return value or use the output of the function without testing the return value.

  The checker covers function from the standard library and other well-known libraries such as the POSIX library or the WinAPI library. Polyspace considers a function as sensitive if the function call is prone to failure because of reasons such as:

  - Exhausted system resources (for example, when allocating resources).
  - Changed privileges or permissions.
  - Tainted sources when reading, writing, or converting data from external sources.
  - Unsupported features despite an existing API.

  Polyspace considers a function a critical sensitive when they perform critical tasks such as:

  - Set privileges (for example, `setuid`)
  - Create a jail (for example, `chroot`)

  - Create a process (for example, fork)
  - Create a thread (for example, pthread_create)
  - Lock or unlock mutex (for example, pthread_mutex_lock)
  - Lock or unlock memory segments (for example, mlock)

  For functions that are not critical, the checker is not flagged if you explicitly ignore the return value by casting it to void. Explicitly ignoring the return value of critical sensitive functions is flagged by Polyspace.

- You call a function that sets errno to indicate error conditions, but do not check errno after the call. For these functions, checking errno is the only reliable way to determine if an error occurred.

  Functions that set errno on errors include:

  - fgetwc, strtol, and wcstol.

    For a comprehensive list of functions, see documentation about errno.
  - POSIXerrno-setting functions such as encrypt and setkey.

#### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Sensitive Function Return Ignored

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>
#include <cstdlib>
#define fatal_error() abort()

void initialize_1() {
    pthread_attr_t attr;
    pthread_attr_init(&attr); //Noncompliant
}

void initialize_2() {
    pthread_attr_t attr;
   (void)pthread_attr_init(&attr); //Compliant
}

void initialize_3() {
    pthread_attr_t attr;
    int result;
    result = pthread_attr_init(&attr); //Compliant
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

```
int read_file_1(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r");
  return 0; //Noncompliant

}
int read_file_2(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r"); //Compliant
  if (in==NULL){
      // Handle error
  }
  return 0;
}
```

This example shows a call to the sensitive functions `pthread_attr_init` and `fmemopen`. Polyspace raises a flag if:

- You implicitly ignore the return of the sensitive function. Explicitly ignoring the output of sensitive functions is not flagged.
- You obtain the return value of a sensitive function but do not test the value before exiting the relevant scope. The violation is raised on the exit statement.

To be compliant, you can explicitly cast their return value to `void` or test the return values to check for errors.

### Critical Function Return Ignored

```
#include <pthread.h>
#include <cstdlib>
#define fatal_error() abort()
extern void *start_routine(void *);

void returnnotchecked_1() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0)); //Noncompliant
    pthread_join(thread_id,  &res); //Noncompliant
}

void returnnotchecked_2() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL); //Compliant
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res); //Compliant
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), the rule checker still raises a violation. The other critical function, `pthread_join`, returns a value that is ignored implicitly.

To be compliant, check the return value of these critical functions to verify the function performed as expected.

### `errno` Not Checked After Call to `strtol`

```
#include<cstdlib>
#include<cerrno>
#include<climits>
#include<iostream>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    long val = strtol(str, &endptr, base); //Noncompliant
    std::cout<<"Return value of strtol() = %ld\n" << val;

    errno = 0;
    long val2 = strtol(str, &endptr, base); //Compliant
    if((val2 == LONG_MIN || val2 == LONG_MAX) && errno == ERANGE) {
         std::cout<<"strtol error";
         exit(EXIT_FAILURE);
    }
    std::cout<<"Return value of strtol() = %ld\n" << val2;
}
```

In the noncompliant example, the return value of `strtol` is used without checking `errno`.

To be compliant, before calling `strtol`, set `errno` to zero . After a call to `strtol`, check the return value for `LONG_MIN` or `LONG_MAX` and `errno` for `ERANGE`.

## Check Information

**Group:** Language Independent Issues
**Category:** Required

# Version History

**Introduced in R2020b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 1-0-1

All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1"

## Description

**Note** To flag instances of code that does not comply with C++03 but complies with more recent versions of C++, specify the option `-cpp-version` as `cpp03`.

### Rule Definition

*All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".*

### Polyspace Implementation

The checker reports compilation errors as detected by a compiler that strictly adheres to the C++03 Standard (ISO/IEC 14882:2003).

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** General
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 2-3-1

Trigraphs shall not be used

## Description

### Rule Definition

*Trigraphs shall not be used.*

### Rationale

You denote trigraphs with two question marks followed by a specific third character (for instance,`'??-'` represents a `'~'` (tilde) character and `'??)'` represents a `']'`). These trigraphs can cause accidental confusion with other uses of two question marks.

For instance, the string

```
"(Date should be in the form ??-??-??)"
```

is transformed to

```
"(Date should be in the form ~~]"
```

but this transformation might not be intended.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Lexical Conventions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 2-5-1

Digraphs should not be used

## Description

### Rule Definition

*Digraphs should not be used.*

### Rationale

Digraphs are a sequence of two characters that are supposed to be treated as a single character. The checker flags use of these digraphs:

- `<%`, indicating {
- `%>`, indicating }
- `<:`, indicating [
- `:>`, indicating ]
- `%:`, indicating #
- `%:%:`, indicating ##

When developing or reviewing code with digraphs, the developer or reviewer can incorrectly consider the digraph as a sequence of separate characters.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Lexical Conventions
**Category:** Advisory

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 2-7-1

The character sequence /* shall not be used within a C-style comment

## Description

### Rule Definition

*The character sequence /* shall not be used within a C-style comment.*

### Rationale

If your code contains a `/*` in a `/* */` comment, it typically means that you have inadvertently commented out code. See the example that follows.

### Polyspace Implementation

You cannot justify a violation of this rule using source code annotations.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `/*` in `/* */` Comment

```
void setup(void);
void foo() {
    /* Initializer functions
     setup();
    /* Step functions */  //Noncompliant
}
```

In this example, the call to `setup()` is commented out because the ending `*/` is omitted, perhaps inadvertently. The checker flags this issue by highlighting the `/*` in the `/* */` comment.

## Check Information

**Group:** Lexical Conventions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 2-7-2

Sections of code shall not be "commented out" using C-style comments

## Description

### Rule Definition

*Sections of code shall not be "commented out" using C-style comments.*

### Rationale

C-style comments enclosed in `/* */` do not support nesting. A comment beginning with `/*` ends at the first `*/` even when the `*/` is intended as the end of a later nested comment. If a section of code that is commented out already contains comments, you can encounter compilation errors (or at least comment out less code than you intend).

Commenting out code is not a good practice. The commented out code can remain out of sync with the surrounding code without causing compilation errors. Later, if you uncomment the code, you can encounter unexpected issues.

Use comments only to explain aspects of the code that are not apparent from the code itself.

### Polyspace Implementation

The checker uses internal heuristics to detect commented out code. For instance, characters such as `#`, `;`, `{` or `}` indicate comments that might potentially contain code. These comments are then evaluated against other metrics to determine the likelihood of code masquerading as comment. For instance, several successive words without a symbol in between reduces this likelihood.

The checker does not flag the following comments even if they contain code:

- Doxygen comments beginning with `/**` or `/*!`.
- Comments that repeat the same symbol several times, for instance, the symbol = here:

  ```
  /* =====================================
   * A comment
   * =====================================*/
  ```
- Comments on the first line of a file.
- Comments that mix the C style (`/* */`) and C++ style (`//`).

The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Code Commented Out With C-Style Comments

```
#include <iostream>
/* class randInt {//Noncompliant
    public:
      int getRandInt();
};
*/

int getRandInt();

/* Function to print random integers*/
void printInteger() {
    /* int val = getRandInt();//Noncompliant
     * val++;
     * std::cout << val;*/
    std::cout << getRandInt();
}
```

This example contains two blocks of commented out code, that constitutes two rule violations.

## Check Information

**Group:** Lexical Conventions
**Category:** Required

# Version History

**Introduced in R2020b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 2-7-3

Sections of code should not be "commented out" using C++-style comments

## Description

### Rule Definition

*Sections of code should not be "commented out" using C++-style comments.*

### Rationale

Commenting out code is not a good practice. The commented out code can remain out of sync with the surrounding code without causing compilation errors. Later, if you uncomment the code, you can encounter unexpected issues.

Use comments only to explain aspects of the code that are not apparent from the code itself.

### Polyspace Implementation

The checker uses internal heuristics to detect commented out code. For instance, characters such as `#`, `;`, `{` or `}` indicate comments that might potentially contain code. These comments are then evaluated against other metrics to determine the likelihood of code masquerading as comment. For instance, several successive words without a symbol in between reduces this likelihood.

The checker does not flag the following comments even if they contain code:

- Doxygen comments beginning with `///` or `//!`.
- Comments that repeat the same symbol several times, for instance, the symbol = here:

```
// =====================================
// A comment
// =====================================*/
```

- Comments on the first line of a file.
- Comments that mix the C style (`/* */`) and C++ style (`//`).

The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Code Commented Out With C++-Style Comments

```
#include <iostream>
int getRandInt();

// Function to print random integers
```

```
void printInteger() {
    // int val = getRandInt();
    // val++;
    // std::cout << val;
    std::cout << getRandInt();
}
```

This example contains a block of commented out code that violates the rule.

## Check Information

**Group:** Lexical Conventions
**Category:** Advisory

# Version History

**Introduced in R2020b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 2-10-1

Different identifiers shall be typographically unambiguous

## Description

### Rule Definition

*Different identifiers shall be typographically unambiguous.*

### Rationale

When you use different identifiers that look typographically similar, you might inadvertently use the incorrect identifier later in your code. Such typographically ambiguous identifiers might lead to bugs that are difficult to diagnose.

Use identifiers that are unambiguously distinct. Avoid using identifiers that differ by a combination of these:

- The use of a lowercase letter instead of an uppercase one, and vice versa.
- The presence or absence of the underscore character.
- The interchange of the letter `O` and the digit `0`.
- The interchange of the letter `I` and the digit `1`.
- The interchange of the letter `I` and the letter `l`.
- The interchange of the letter `S` and the digit `5`.
- The interchange of the letter `Z` and the digit `2`.
- The interchange of the letter `n` and the letter `h`.
- The interchange of the letter `B` and the digit `8`.
- The interchange of the letters `rn` and the letter `m`.

### Polyspace Implementation

Polyspace reports a defect if two identifiers differ by a combination of ambiguous characters mentioned in the preceding list. The rule checker does not consider the fully qualified names of variables when checking this rule.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val;  //Non-compliant
    int id2_numval;
    int id2_numVal;   //Non-compliant

    int id3_lvalue;
    int id3_Ivalue;   //Non-compliant

    int id4_xyZ;
    int id4_xy2;      //Non-compliant

    int id5_zerO;
    int id5_zer0;     //Non-compliant

    int id6_rn;
    int id6_m;        //Non-compliant
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

## Check Information

**Group:** Lexical Conventions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 2-10-2

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope

## Description

### Rule Definition

*Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.*

### Rationale

The rule flags situations where the same identifier name is used in two variable declarations, one in an outer scope and the other in an inner scope.

```
int var;
...
{
...
  int var;
...
}
```

All uses of the name in the inner scope refers to the variable declared in the inner scope. However, a developer or code reviewer can incorrectly assume that the usage refers to the variable declared in the outer scope.

### Polyspace Implementation

The rule checker flags all cases of variable shadowing including when:

- The same identifier name is used in an outer and inner named namespace.
- The same name is used for a class data member and a variable outside the class.
- The same name is used for a method in a base and derived class.

To exclude these cases, switch to the more modern standard AUTOSAR C++14 and check for the rule `AUTOSAR C++14 Rule A2-10-1`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Local Variable Hiding Global Variable

```
int varInit = 1;

void doSomething(void);

void step(void) {
    int varInit = 0; //Noncompliant
    if(varInit)
       doSomething();
}
```

In this example, `varInit` defined in `func` hides the global variable `varInit`. The `if` condition refers to the local `varInit` and the block is unreachable, but you might expect otherwise.

### Loop Index Hiding Variable Outside Loop

```
void runSomeCheck(int);

void checkMatrix(int dim1, int dim2) {
  for(int index = 0; index < dim1; index++) {
      for(int index = 0; index < dim2; index++) { // Noncompliant
          runSomeCheck(index);
      }
  }
}
```

In this example, the variable `index` defined in the inner `for` loop hides the variable with the same name in the outer loop.

## Check Information

**Group:** Lexical Conventions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 2-10-3

A typedef name (including qualification, if any) shall be a unique identifier

## Description

### Rule Definition

*A typedef name (including qualification, if any) shall be a unique identifier.*

### Rationale

When you use reuse `typedef` names, the code becomes confusing and difficult to maintain. You might use the wrong identifier and introduce bugs that are difficult to fix.

### Polyspace Implementation

The rule reports a violation on identifier declarations where the identifier name is the same as a previously declared typedef name. The checker does not flag situations where the conflicting names occur in different namespaces.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Typedef Name Conflicting with Other Identifiers

```
namespace NS1 {
    typedef int WIDTH;
}

namespace NS2 {
    float WIDTH; //Compliant
}

void f1() {
    typedef int TYPE;
}

void f2() {
    float TYPE; //Noncompliant
}
```

In this example, the declaration of `TYPE` in `f2()` conflicts with a typedef declaration in `f1()`.

The checker does not flag the redeclaration of `WIDTH` because the two declarations belong to different namespaces.

## Check Information

**Group:** Lexical Conventions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 2-10-4

A class, union or enum name (including qualification, if any) shall be a unique identifier

## Description

### Rule Definition

*A class, union or enum name (including qualification, if any) shall be a unique identifier.*

### Rationale

Using nonunique identifiers makes the code confusing and difficult to maintain. You might use the wrong identifier in your code and introduce bugs that are difficult to fix.

### Polyspace Implementation

The rule flags identifier declarations where the identifier name is the same as a previously declared class, union or typedef name. The checker does not flag situations where the conflicting names occur in different namespaces.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Typedef Name Conflicting with Other Identifiers

```
void f1() {
    class floatVar {};
}

void f2() {
    float floatVar; //Noncompliant
}
```

In this example, the declaration of `floatVar` in `f2()` conflicts with a class declaration in `f1()`.

## Check Information

**Group:** Lexical Conventions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 2-10-5

The identifier name of a non-member object or function with static storage duration should not be reused

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis . See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*The identifier name of a non-member object or function with static storage duration should not be reused.*

### Rationale

Reusing the name of an identifier with static storage duration makes the code confusing and difficult to maintain. You might use the wrong identifier in your code and introduce bugs that are difficult to fix.

The rule applies even if the identifiers belong to different namespaces because the reuse leaves the chance that you mistake one identifier for the other.

### Polyspace Implementation

The rule flags situations where the name of an identifier with static storage duration is reused. The rule checker flags redefined functions only when there is a declaration.

The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` or `extern` functions
- Uncalled and undefined local functions
- Unused types and variables

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Reuse of Identifiers in Different Namespaces

```
namespace NS1 {
    static int WIDTH;
}

namespace NS2 {
    float WIDTH; //Noncompliant
}
```

In this example, the identifier name `WIDTH` is reused in the two namespaces `NS1` and `NS2`.

## Check Information

**Group:** Lexical Conventions
**Category:** Advisory

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 2-10-6

If an identifier refers to a type, it shall not also refer to an object or a function in the same scope

## Description

### Rule Definition

*If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.*

### Rationale

To maintain compatibility with C code, the C++ standard allows you to use the same name for a type and an object or function. However, the name reuse can cause confusion during development or code review.

### Polyspace Implementation

Polyspace reports a violation of this rule if you use an identifier both as a type name and as a function or object name. If the identifier is a function name and the function is both declared and defined, then the violation is reported once.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Reuse of Name for Type and Object

```
struct vector{
    int x;
    int y;
    int z;
}vector; //Noncompliant
```

In this example, the name `vector` is used both for the structured data type and for an object of that type.

## Check Information

**Group:** Lexical Conventions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 2-13-1

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used

## Description

### Rule Definition

*Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.*

### Rationale

Escape sequences are certain special characters represented in string and character literals. They are written with a backslash (\) followed by a character.

The C++ Standard (ISO/IEC 14882:2003, Sec. 2.13.2) defines a list of escape sequences. See Escape Sequences. Use of escape sequences (backslash followed by character) outside that list leads to undefined behavior.

### Additional Message in Report

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

`\`*`char`* is not an ISO/IEC escape sequence.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Incorrect Escape Sequences

```
void func () {
  const char a[2] = "\k"; //Noncompliant
  const char b[2] = "\b"; //Compliant
}
```

In this example, `\k` is not a recognized escape sequence.

## Check Information

**Group:** Lexical Conventions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 2-13-2

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used

## Description

### Rule Definition

*Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.*

### Rationale

Octal constants are denoted by a leading zero. A developer or code reviewer can mistake an octal constant as a decimal constant with a redundant leading zero.

Octal escape sequences beginning with \ can also cause confusion. Inadvertently introducing an 8 or 9 in the digit sequence after \ breaks the escape sequence and introduces a new digit. A developer or code reviewer can ignore this issue and continue to treat the escape sequence as one digit.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Octal Constants and Octal Escape Sequences

```
void func(void) {
  int busData[6];

  busData[0] = 100;
  busData[1] = 108;
  busData[2] = 052;      //Noncompliant
  busData[3] = 071;      //Noncompliant
  busData[4] = '\109';   //Noncompliant
  busData[5] = '\100';   //Noncompliant

}
```

The checker flags all octal constants (other than zero) and all octal escape sequences (other than `\0`).

In this example:

- The octal escape sequence contains the digit 9, which is not an octal digit. This escape sequence has implementation-defined behavior.
- The octal escape sequence `\100` represents the number 64, but the rule checker forbids this use.

## Check Information

**Group:** Lexical Conventions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 2-13-3

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type

## Description

### Rule Definition

*A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.*

### Rationale

The signedness of a constant is determined from:

- Value of the constant.
- Base of the constant: octal, decimal or hexadecimal.
- Size of the various types.
- Any suffixes used.

Unless you use a suffix u or U, another developer looking at your code cannot determine easily whether a constant is signed or unsigned.

### Polyspace Implementation

Polyspace reports a violation of this rule if an unsigned octal or hexadecimal constant does not have the suffix U.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unsigned Integer Literal Missing Suffix

In this example, the unsigned integer literal 0x8000 does not have the U suffix. This literal is signed in a 32-bit environment but unsigned in a 16-bit environment. In a 16-bit environment, this code violates the rule. To see this rule violation, set -target to a 16-bit processor, such as c-167.

```
typedef unsigned int uint;
void foo ( )
{
uint a = 0U; // Compliant
uint b = 0x8000; //Noncompliant for a 16-bit processor
}
```

## Check Information

**Group:** Lexical Conventions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 2-13-4

Literal suffixes shall be upper case

## Description

### Rule Definition

*Literal suffixes shall be upper case.*

### Rationale

Literal constants can end with the letter `l` (el). Enforcing literal suffixes to be upper case removes potential confusion between the letter `l` and the digit `1`.

For consistency, use upper case constants for other suffixes such as `U` (unsigned) and `F` (float).

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Literal Constants with Lower Case Suffix

```
const int a = 0l; //Noncompliant
const int b = 0L; //Compliant
```

In this example, both `a` and `b` are assigned the same literal constant. However, from a quick glance, one can mistakenly assume that `a` is assigned the value `01` (octal one).

## Check Information

**Group:** Lexical Conventions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 2-13-5

Narrow and wide string literals shall not be concatenated

## Description

### Rule Definition

*Narrow and wide string literals shall not be concatenated.*

### Rationale

Narrow string literals are enclosed in double quotes without a prefix. Wide string literals are enclosed in double quotes with a prefix `L` outside the quotes. See string literals.

Concatenation of narrow and wide string literals can lead to undefined behavior.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Concatenation of Narrow and Wide String Literals

```
char array[] = "Hello" "World";
wchar_t w_array[] = L"Hello" L"World";
wchar_t mixed[] = "Hello" L"World"; //Noncompliant
```

In this example, in the initialization of the array `mixed`, the narrow string literal `"Hello"` is concatenated with the wide string literal `L"World"`.

## Check Information

**Group:** Lexical Conventions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 3-1-1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule

## Description

### Rule Definition

*It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.*

### Rationale

If a header file with variable or function definitions appears in multiple inclusion paths, the header file violates the One Definition Rule possibly leading to unpredictable behavior. For instance, a source file includes the header file `include.h` and another header file, which also includes `include.h`.

### Polyspace Implementation

The rule checker flags variable and function definitions in header files.

Polyspace reports violation of this rule in header files. In a nonheader source file, violation of this rule is not reported.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Variable and Function Definitions in Header Files

In this example, Polyspace reports violations of this rule when variables and function definitions are placed in the header file `header.hpp`.

```
// header.hpp
#include <cstdint>
void F1();        // Compliant
extern void F2(); // Compliant
void F3()         // Noncompliant
{
}
static inline void F4()
{
} // Compliant
template <typename T>
void F5(T) // Compliant
{
}
std::int32_t a;                        // Noncompliant
extern std::int32_t b;                 // Compliant
constexpr static std::int32_t c = 10; // Compliant
namespace ns
{
    constexpr static std::int32_t d = 100; // Compliant
    const static std::int32_t e = 50;      // Compliant
    static std::int32_t f;                 // Noncompliant
    static void F6() noexcept;             // Noncompliant
}
```

```
//source.cpp
#include "header.hpp"
void foo()
{
    //...
}
```

## Check Information

**Group:** Basic Concepts
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 3-1-2

Functions shall not be declared at block scope

## Description

### Rule Definition

*Functions shall not be declared at block scope.*

### Rationale

It is a good practice to place all declarations at the namespace level.

Additionally, if you declare a function at block scope, it is often not clear if the statement is a function declaration or an object declaration with a call to the constructor.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Function Declarations at Block Scope

```
class A {
};

void b1() {
    void func(); //Noncompliant
    A a();   //Noncompliant
}
```

In this example, the declarations of `func` and `a` are in the block scope of `b1`.

The second function declaration can cause confusion because it is not clear if `a` is a function that returns an object of type `A` or `a` is itself an object of type `A`.

## Check Information

**Group:** Basic Concepts
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 3-1-3

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization

## Description

### Rule Definition

*When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.*

### Rationale

Though you can declare an incomplete array type and later complete the type, specifying the array size during the first declaration makes the subsequent array access less error-prone.

### Additional Message in Report

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Size of array `arrayName` should be explicitly stated.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Array Size Unspecified During Declaration

```
int array[10];
extern int array2[]; //Noncompliant
int array3[]= {0,1,2};
extern int array4[10];
```

In the declaration of `array2`, the array size is unspecified.

## Check Information

**Group:** Basic Concepts
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 3-2-1

All declarations of an object or function shall have compatible types

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis . See “Checkers Deactivated in Polyspace as You Code Analysis”.*

### Rule Definition

*All declarations of an object or function shall have compatible types.*

### Rationale

If the declarations of an object or function in two different translation units have incompatible types, the behavior is undefined.

### Polyspace Implementation

Polyspace considers two types to be compatible if they have the same size and signedness in the environment that you use. The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` or `extern` functions
- Uncalled and undefined local functions
- Unused types and variables

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Compatible and Incompatible Definitions in Two Files

**file1.cpp**

```
typedef          char         char_t;
typedef signed   short        int16_t;
typedef signed   long         int64_t;

namespace bar {
    int64_t a;
    int16_t c;

};
```

**file2.cpp**

```
typedef          char         char_t;
typedef signed   int          int32_t;

namespace bar {
    extern char_t c;// Noncompliant
    extern int32_t a;
    void foo(void){
        ++a;
        ++c;
    }
};
```

In this example, the variable `bar::c` is defined as a `char` in `file2.cpp` and as a `signed short` in `file1.cpp`. In the target processor i386, the size of these types are not equal. Polyspace flags the definition of `bar::c`.

The variable `bar::a` is defined as a `long` in `file1.cpp` and as an `int` in `file2.cpp`. In the target processor i386, both `int` and `long` has a size of 32 bits. Because the definitions of `bar::a` is compatible in both files, Polyspace does not raise a flag.

## Check Information

**Group:** Basic Concepts
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 3-2-2

The One Definition Rule shall not be violated

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis . See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*The One Definition Rule shall not be violated.*

### Rationale

Violations of the One Definition Rule leads to undefined behavior.

### Polyspace Implementation

The checker flags situations where the same function or object has multiple definitions and the definitions differ by some token. The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` or `extern` functions
- Uncalled and undefined local functions
- Unused types and variables

### Additional Message in Report

The One Definition Rule shall not be violated.

Declaration of class *`className`* violates the One Definition Rule:

it conflicts with other declaration (*`fileNamelineNumber`*).

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Different Tokens in Same Type Definition

This example uses two files:

- `file1.cpp`:

```
typedef struct S //Noncompliant
{
   int x;
   int y;
}S;
void foo(S& s){
//...
}
```

- `file2.cpp`:

```
typedef struct S
{
   int y;
   int x;
}S ;
void bar(S& s){
//...
}
```

In this example, both `file1.cpp` and `file2.cpp` define the structure `S`. However, the definitions switch the order of the structure fields.

## Check Information

**Group:** Basic Concepts
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 3-2-3

A type, object or function that is used in multiple translation units shall be declared in one and only one file

## Description

### Rule Definition

*A type, object or function that is used in multiple translation units shall be declared in one and only one file.*

### Rationale

If you declare an identifier in a header file, you can include the header file in any translation unit where the identifier is defined or used. In this way, you ensure consistency between:

- The declaration and the definition.
- The declarations in different translation units.

The rule enforces the practice of declaring external objects or functions in header files.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Basic Concepts
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 3-2-4

An identifier with external linkage shall have exactly one definition

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis . See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*An identifier with external linkage shall have exactly one definition.*

### Rationale

If an identifier has multiple definitions or no definitions, it can lead to undefined behavior.

### Polyspace Implementation

The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` or `extern` functions
- Uncalled and undefined local functions
- Unused types and variables

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Multiple Definitions of Identifier

This example uses two files:

- `file1.cpp`:

```
typedef signed   int          int32_t;

namespace NS {
    extern int32_t a;

    void foo(){
        a = 0;

    }
};
```

- `file2.cpp`:

```
typedef signed   int          int32_t;
typedef signed   long long         int64_t;

namespace NS {
    extern int64_t a; //Noncompliant
    void bar(){
        ++a;

    }
};
```

The same identifier `a` is defined in both files.

## Check Information

**Group:** Basic Concepts
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 3-3-1

Objects or functions with external linkage shall be declared in a header file

## Description

### Rule Definition

*Objects or functions with external linkage shall be declared in a header file.*

### Rationale

If you declare a function or object in a header file, it is clear that the function or object is meant to be accessed in multiple translation units. If you intend to access the function or object from a single translation unit, declare it `static` or in an unnamed namespace.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Declaration in Header File Missing

This example uses two files:

- `decls.h`:

  ```
  extern int x;
  ```
- `file.cpp`:

  ```
  #include "decls.h"

  int x = 0;
  int y = 0; //Noncompliant
  static int z = 0;
  ```

In this example, the variable `x` is declared in a header file but the variable `y` is not. The variable `z` is also not declared in a header file but it is declared with the `static` specifier and does not have external linkage.

## Check Information

**Group:** Basic Concepts
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 3-3-2

If a function has internal linkage then all re-declarations shall include the static storage class specifier

## Description

### Rule Definition

*If a function has internal linkage then all re-declarations shall include the static storage class specifier.*

### Rationale

If a function declaration has the `static` storage class specifier, it has internal linkage. Subsequent redeclarations of the function have internal linkage even without the `static` specifier.

However, if you do not specify the `static` keyword explicitly, it is not immediately clear from a declaration whether the function has internal linkage.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Missing `static` Specifier from Redeclaration

```
static void func1 ();
static void func2 ();

void func1() {}  //Noncompliant
static void func2() {}
```

In this example, the function `func1` is declared `static` but defined without the `static` specifier.

## Check Information

**Group:** Basic Concepts
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 3-4-1

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility

## Description

### Rule Definition

*An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.*

### Rationale

Defining variables with the minimum possible block scope reduces the possibility that they might later be accessed unintentionally.

For instance, if an object is meant to be accessed in one function only, declare the object local to the function.

### Polyspace Implementation

The rule checker determines if an object is used in one block only. If the object is used in one block but defined outside the block, the checker raises a violation.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Global Variable in Single Function

```
static int countReset; //Noncompliant

volatile int check;

void increaseCount() {
    int count = countReset;
    while(check%2) {
     count++;
    }
}
```

In this example, the variable `countReset` is declared global used in one function only. A compliant solution declares the variable local to the function to reduce its visibility.

## Check Information

**Group:** Basic Concepts
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 3-9-1

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations

## Description

### Rule Definition

*The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.*

### Rationale

If a redeclaration is not token-for-token identical to the previous declaration, it is not clear from visual inspection which object or function is being redeclared.

### Polyspace Implementation

The rule checker compares the current declaration with the last seen declaration.

### Additional Message in Report

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Variable *varName* is not compatible with previous declaration.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Identical Declarations That Do Not Match Token for Token

```
typedef int* intptr;

int* map;
extern intptr map; //Noncompliant

intptr table;
extern intptr table; //Compliant
```

In this example, the variable `map` is declared twice. The second declaration uses a `typedef` which resolves to the type of the first declaration. Because of the `typedef`, the second declaration is not token-for-token identical to the first.

## Check Information

**Group:** Basic Concepts

**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 3-9-2

typedefs that indicate size and signedness should be used in place of the basic numerical types

## Description

### Rule Definition

*typedefs that indicate size and signedness should be used in place of the basic numerical types.*

### Rationale

When the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object.

### Polyspace Implementation

The rule checker does not raise violations in templates that are not instantiated.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Direct Use of Basic Numerical Types

```
typedef unsigned int uint32_t;

unsigned int x = 0;       //Noncompliant
uint32_t y = 0;  //Compliant
```

In this example, the declaration of `x` is noncompliant because it uses the basic type `int` directly.

## Check Information

**Group:** Basic Concepts
**Category:** Advisory

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 3-9-3

The underlying bit representations of floating-point values shall not be used

## Description

### Rule Definition

*The underlying bit representations of floating-point values shall not be used.*

### Rationale

The underlying bit representations of floating point values vary across compilers. If you directly use the underlying representation of floating point values, your program is not portable across implementations.

### Polyspace Implementation

The rule checker flags conversions from pointers to floating point types into pointers to integer types, and vice versa.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Using Underlying Representation of Floating-Point Values

```
float fabs2(float f) {
    unsigned int* ptr = reinterpret_cast <unsigned int*> (&f); //Noncompliant
    *(ptr + 3) &= 0x7f;
    return f;
}
```

In this example, the `reinterpret_cast` attempts to cast a floating-point value to an integer and access the underlying bit representation of the floating point value.

## Check Information

**Group:** Basic Concepts
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 4-5-1

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator

## Description

### Rule Definition

*Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.*

### Rationale

Operators other than the ones mentioned in the rule do not produce meaningful results with `bool` operands. Use of `bool` operands with these operators can indicate programming errors. For instance, you intended to use the logical operator `||` but used the bitwise operator `|` instead.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Compliant and Noncompliant Uses of `bool` Operands

```
void boolOperations() {
    bool lhs = true;
    bool rhs = false;

    int res;

    if(lhs & rhs) {} //Noncompliant
    if(lhs < rhs) {} //Noncompliant
    if(~rhs) {}      //Noncompliant
    if(lhs ^ rhs) {} //Noncompliant
    if(lhs == rhs) {} //Compliant
    if(!rhs) {}       //Compliant
    res = lhs? -1:1;  //Compliant
}
```

In this example, `bool` operands do not violate the rule when used with the `==`, `!` and the `?` operators.

## Check Information

**Group:** Standard Conversions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 4-5-2

Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=

## Description

### Rule Definition

*Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Standard Conversions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 4-5-3

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator N

## Description

### Rule Definition

*Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N*

### Rationale

The C++03 Standard only requires that the characters `'0'` to `'9'` have consecutive values. Other characters do not have well-defined values. If you use these characters in operations other than the ones mentioned in the rule, you implicitly use their underlying values and might see unexpected results.

### Additional Message in Report

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Compliant and Noncompliant Uses of Character Operands

```
void charManipulations (char ch) {

    char initChar = 'a'; //Compliant
    char finalChar = 'z'; //Compliant

    if(ch == initChar) {} //Compliant
    if( (ch >= initChar) && (ch <= finalChar)) {} //Noncompliant
    else if( (ch >= '0') && (ch <= '9') ) {} //Compliant by exception
}
```

In this example, character operands do not violate the rule when used with the = and == operators. Character operands can also be used with relational operators as long as the comparison is performed with the digits `'0'` to `'9'`.

## Check Information

**Group:** Standard Conversions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 4-10-1

NULL shall not be used as an integer value

## Description

### Rule Definition

*NULL shall not be used as an integer value.*

### Rationale

In C++, you can use the literals 0 and NULL as both an integer and a null pointer constant. However, use of 0 as a null pointer constant or NULL as an integer can cause developer confusion.

This rule restricts the use of NULL to null pointer constants. `MISRA C++:2008 Rule 4-10-2` restricts the use of the literal 0 to integers.

### Polyspace Implementation

The checker flags assignment of NULL to an integer variable or binary operations involving NULL and an integer. Assignments can be direct or indirect such as passing NULL as integer argument to a function.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Compliant and Noncompliant Uses of `NULL`

```
#include <cstddef>

void checkInteger(int);
void checkPointer(int *);

void main() {
    checkInteger(NULL); //Noncompliant
    checkPointer(NULL); //Compliant
}
```

In this example, the use of `NULL` as argument to the `checkInteger` function is noncompliant because the function expects an `int` argument.

## Check Information

**Group:** Standard Conversions
**Category:** Required

## Version History

**Introduced in R2018a**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 4-10-2

Literal zero (0) shall not be used as the null-pointer-constant

## Description

### Rule Definition

*Literal zero (0) shall not be used as the null-pointer-constant.*

### Rationale

In C++, you can use the literals 0 and NULL as both an integer and a null pointer constant. However, use of 0 as a null pointer constant or NULL as an integer can cause developer confusion.

This rule restricts the use of the literal 0 to integers. `MISRA C++:2008 Rule 4-10-1` restricts the use of NULL to null pointer constants.

### Polyspace Implementation

The checker flags assignment of 0 to a pointer variable or binary operations involving 0 and a pointer. Assignments can be direct or indirect such as passing 0 as pointer argument to a function.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Compliant and Noncompliant Uses of Literal 0

```
#include <cstddef>

void checkInteger(int);
void checkPointer(int *);

void main() {
    checkInteger(0); //Compliant
    checkPointer(0); //Noncompliant
}
```

In this example, the use of 0 as argument to the `checkPointer` function is noncompliant because the function expects an `int *` argument.

## Check Information

**Group:** Standard Conversions
**Category:** Required

# Version History

**Introduced in R2018a**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-1

The value of an expression shall be the same under any order of evaluation that the standard permits

## Description

### Rule Definition

*The value of an expression shall be the same under any order of evaluation that the standard permits.*

### Rationale

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

### Polyspace Implementation

Polyspace raises a violation if an expression satisfies any of these conditions:

- The same variable is modified more than once in the expression or it is both read and written.
- The expression allows more than one order of evaluation.
- The expression contains a single `volatile` object that occurs multiple times.
- The expression contains more than one `volatile` object.

Because `volatile` objects can change their value at anytime, an expression containing multiple `volatile` variables or multiple instances of the same `volatile` variable might have different results depending on the order of evaluation.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])
void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         // Compliant
    COPY_ELEMENT (i++);    // // Non-compliant
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

### Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
    f ( i++, i );                 // Noncompliant
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

### Multiple `volatile` Objects in an Expression

```
volatile int a, b;
int mathOp(int x, int y);

int foo(void){
    int temp = mathOp(5,a) + mathOp(6,b);//Noncompliant
    return temp * mathOp(a,a);//Noncompliant
}
```

In this example, this rule is violated twice.

- The declaration of `temp` uses two `volatile` objects in the expression. Because the value of `volatile` objects might change at any time, the expression might evaluate to different values depending on the order of evaluation. Polyspace flags the second `volatile` object in the expression.
- The `return` statement uses the same `volatile` object twice. Because the expression might have different results depending on the order of evaluation, Polyspace raises this defect.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-2

Limited dependence should be placed on C++ operator precedence rules in expressions

## Description

### Rule Definition

*Limited dependence should be placed on C++ operator precedence rules in expressions.*

### Rationale

Use parentheses to clearly indicate the order of evaluation.

Depending on operator precedence can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:
  - In the operation `*p++`, it is possible that you expect the dereferenced value to be incremented. However, the pointer `p` is incremented before the dereference.
  - In the operation `(x == y | z)`, it is possible that you expect `x` to be compared with `y | z`. However, the `==` operation happens before the `|` operation.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Evaluation Order Dependent on Operator Precedence Rules

```
#include <cstdio>

void showbits(unsigned int x) {
    for(int i = (sizeof(int) * 8) - 1; i >= 0; i--) {
       (x & 1u << i) ? putchar('1') : putchar('0'); // Noncompliant
    }
    printf("\n");
}
```

In this example, the checker flags the operation `x & 1u << i` because the statement relies on operator precedence rules for the `<<` operation to happen before the `&` operation. If this is the intended order, the operation can be rewritten as `x & (1u << i)`.

## Check Information

**Group:** Expressions
**Category:** Advisory

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-3

A cvalue expression shall not be implicitly converted to a different underlying type

## Description

### Rule Definition

*A cvalue expression shall not be implicitly converted to a different underlying type.*

### Rationale

This rule ensures that the result of the expression does not overflow when converted to a different type.

### Polyspace Implementation

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

The underlying data type of a cvalue expression is the widest of operand data types in the expression. For instance, if you add two variables, one of type `int8_t` (`typedef` for `char`) and another of type `int32_t` (`typedef` for `int`), the addition has underlying type `int32_t`. If you assign the sum to a variable of type `int8_t`, the rule is violated.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Implicit Conversion of Cvalue Expression

```
#include<cstdint>

void func ( )
  {
    int32_t s32;
    int8_t s8;
    s32 = s8 + s8; //Noncompliant
    s32 = s32 + s8; //Compliant
  }
```

In this example, the rule is violated when two variables of type `int8_t` are added and the result is assigned to a variable of type `int32_t`. The underlying type of the addition does not take into account the integer promotion involved and is simply the widest of operand data types, in this case, `int8_t`.

The rule is not violated if one of the operands has type `int32_t` and the result is assigned to a variable of type `int32_t`. In this case, the underlying data type of the addition is the same as the type of the variable to which the result is assigned.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 5-0-4

An implicit integral conversion shall not change the signedness of the underlying type

## Description

### Rule Definition

*An implicit integral conversion shall not change the signedness of the underlying type.*

### Rationale

Some conversions from signed to unsigned data types can lead to implementation-defined behavior. You can see unexpected results from the conversion.

### Polyspace Implementation

The checker flags implicit conversions from a signed to an unsigned integer data type or vice versa.

The checker assumes that `ptrdiff_t` is a signed integer.

### Additional Message in Report

An implicit integral conversion shall not change the signedness of the underlying type.

Implicit conversion of one of the binary + operands whose underlying types are *`typename_1`* and *`typename_2`*.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Implicit Conversions that Change Signedness

```
typedef char int8_t;
typedef unsigned char uint8_t;

void func()
  {
    int8_t s8;
    uint8_t u8;

    s8 = u8; //Noncompliant
    u8 = s8 + u8; //Noncompliant
    u8 = static_cast< uint8_t > ( s8 ) + u8; //Compliant
}
```

In this example, the rule is violated when a variable with a variable with signed data type is implicitly converted to a variable with unsigned data type or vice versa. If the conversion is explicit, as in the preceding example, the rule violation does not occur.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-5

There shall be no implicit floating-integral conversions

## Description

### Rule Definition

*There shall be no implicit floating-integral conversions.*

### Rationale

If you convert from a floating point to an integer type, you lose information. Unless you explicitly cast from floating point to an integer type, it is not clear whether the loss of information is intended. Additionally, if the floating-point value cannot be represented in the integer type, the behavior is undefined.

Conversion from an integer to floating-point type can result in an inexact representation of the value. The error from conversion can accumulate over later operations and lead to unexpected results.

### Polyspace Implementation

The checker flags implicit conversions between floating-point types (`float` and `double`) and integer types (`short`, `int`, etc.).

This rule takes precedence over 5-0-4 and 5-0-6 if they apply at the same time.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Conversion Between Floating Point and Integer Types

```
typedef signed int int32_t;
typedef float float32_t;

void func ( )
  {
    float32_t f32;
    int32_t   s32;
    s32 = f32;   //Noncompliant
    f32 = s32;   //Noncompliant
    f32 = static_cast< float32_t > ( s32 ); //Compliant
  }
```

In this example, the rule is violated when a floating-point type is *implicitly* converted to an integer type. The violation does not occur if the conversion is explicit.

## Check Information

**Group:** Expressions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-6

An implicit integral or floating-point conversion shall not reduce the size of the underlying type

## Description

### Rule Definition

*An implicit integral or floating-point conversion shall not reduce the size of the underlying type.*

### Rationale

A conversion that reduces the size of the underlying type can result in loss of information. Unless you explicitly cast to the narrower type, it is not clear whether the loss of information is intended.

### Polyspace Implementation

The checker flags implicit conversions that reduce the size of a type.

If the conversion is to a narrower integer with a different sign, then rule 5-0-4 takes precedence over rule 5-0-6. Only rule 5-0-4 is shown.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Conversion That Reduces Size of Type

```
typedef signed short int16_t;
typedef signed int int32_t;

void func ( )
  {
    int16_t   s16;;
    int32_t   s32;
    s16 = s32;   //Noncompliant
    s16 = static_cast< int16_t > ( s32 ); //Compliant
  }
```

In this example, the rule is violated when a type is *implicitly* converted to a narrower type. The violation does not occur if the conversion is explicit.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-7

There shall be no explicit floating-integral conversions of a cvalue expression

## Description

### Rule Definition

*There shall be no explicit floating-integral conversions of a cvalue expression.*

### Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression). For instance, in this example, the result of an integer division is then cast to a floating-point type.

```
short num;
short den;
float res;
res= static_cast<float> (num/den);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a floating-point division because of the later cast.

### Additional Message in Report

There shall be no explicit floating-integral conversions of a cvalue expression.

Complex expression of underlying type *typeBeforeConversion* may only be cast to narrower integer type of same signedness, however the destination type is *typeAfterconversion*.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Conversion of Division Result from Integer to Floating Point

```
void func() {
     short num;
     short den;
     short res_short;
     float res_float;

     res_float = static_cast<float> (num/den); //Noncompliant

     res_short = num/den;
     res_float = static_cast<float> (res_short); //Compliant
}
```

In this example, the first cast on the division result violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the expression is evaluated with an underlying type `float`.
- The second cast makes it clear that the expression is evaluated with the underlying type `short`. The result is then cast to the type `float`.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-8

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression

## Description

### Rule Definition

*An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.*

### Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression). For instance, in this example, the sum of two `short` operands is cast to the wider type `int`.

```
short op1;
short op2;
int res;
res= static_cast<int> (op1 + op2);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a sum with the underlying type `int` because of the later cast.

### Additional Message in Report

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Complex expression of underlying type *`typeBeforeConversion`* may only be cast to narrower integer type of same signedness, however the destination type is *`typeAfterconversion`*.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Conversion of Sum to Wider Integer Type

```
void func() {
     short op1;
     short op2;
     int res;

     res = static_cast<int> (op1 + op2); //Noncompliant
```

```
    res = static_cast<int> (op1) + op2; //Compliant

}
```

In this example, the first cast on the sum violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the sum is evaluated with an underlying type `int`.
- The second cast first converts one of the operands to `int` so that the sum is actually evaluated with the underlying type `int`.

### Check Information

**Group:** Expressions
**Category:** Required

## Version History

**Introduced in R2013b**

### See Also

`Check MISRA C++:2008 (-misra-cpp)`

#### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-9

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression

## Description

### Rule Definition

*An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.*

### Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression).. For instance, in this example, the sum of two `unsigned int` operands is cast to the type `int`.

```
unsigned int op1;
unsigned int op2;
int res;
res= static_cast<int> (op1 + op2);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a sum with the underlying type `int` because of the later cast.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Conversion of Sum to Wider Integer Type

```
typedef int int32_t;
typedef unsigned int uint32_t;

void func() {
     uint32_t op1;
     uint32_t op2;
     int32_t res;

     res = static_cast<int32_t> (op1 + op2); //Noncompliant
     res = static_cast<int32_t> (op1) +
           static_cast<int32_t> (op2); //Compliant

}
```

In this example, the first cast on the sum violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the sum is evaluated with an underlying type `int32_t`.
- The second cast first converts each of the operands to `int32_t` so that the sum is actually evaluated with the underlying type `int32_t`.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-10

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand

## Description

### Rule Definition

*If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.*

### Rationale

When the bitwise operators ~ and << are applied to small integer types, such as unsigned short and unsigned char, the operations are preceded by integral promotion. That is, the small integer types are first promoted to a larger integer type, and then the operation takes place. The result of these bitwise operation might contain unexpected higher order bits. For instance:

```
uint8_t var = 0x5aU;
uint8_t result = (~var)>>4;
```

The binary representation of `var` is `0101 1010` and that of `~var` is `1010 0101`. You might expect that `result` is `0000 1010`. Because `var` is promoted to a larger integer before `~var` is calculated, result becomes `1111 1010`. The higher order bits might be unexpected. The results of such operations might depend on the size of `int` in your implementation.

To avoid confusion and unexpected errors, cast the result of the bitwise ~ and >> operators back to the underlying type of the operands before using the results. For instance:

```
uint8_t var = 0x5aU;
uint8_t result = (static_cat<unit8_t>(~var))>>4;
```

The binary representation of `result` in this case is `0000 1010`, which is the expected value.

As an exception, casting is not required if you apply these bitwise operators on short integer types, and then immediately assign the result to an object of the same underlying type. For instance, the value of result in this case is `0000 1010` without requiring a cast.

```
uint8_t var = 0x5aU;
unit8_t result = ~var; // No higher order bits
                       // due to implicit conversion
uint8_t result = results>>4;
```

### Polyspace Implementation

Polyspace flags the use of the bitwise ~ and >> operators if all of these conditions are true:

- The operators are used on an `unsigned short` or `unsigned char` operand.
- The result of the operation is not immediately assigned to an object that has the same underlying type as the operand.
- The result is used without being cast to the underlying type of the operand.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Cast Results of ~ and << Operators to the Operand Type When the Operand Is Small Integer Type

```
#include<cstdint>
void foo(){
    uint8_t var = 0x5aU;
    uint8_t result;
    result = ( ~var ) >> 4; // Non-compliant
    result = static_cast<uint8_t>(( ~var )) >> 4; // Compliant
    uint8_t cbe = ~var;//Compliant by Exception
}
```

In this example, Polyspace flags the use of ~ on the small integer `var`. The ~ operator is flagged because:

- It operates on an unsigned short integer `var`.
- The result of the operator is used in an expression without casting ~`var` to `uint8_t`.

When the result of ~ operator is cast to `unit8_t`, the use is compliant with this rule. When the result of ~ is immediately assigned to a `unit8_t` variable, the use is compliant to this rule by exception.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 5-0-11

The plain char type shall only be used for the storage and use of character values

## Description

### Rule Definition

*The plain char type shall only be used for the storage and use of character values.*

### Rationale

The signedness of plain `char` is implementation-defined. Because its sign is not well defined, the plain `char` type is not suitable to be used with numeric values. Use plain `char` for the storage and use of character values.

### Polyspace Implementation

The checker raises a violation when a value of signed or unsigned integer type is implicitly converted to the plain char type.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Using Plain `char` For Numeric Data

```
#include<stdint.h>
typedef  char char_t;
void foo(){
char a = 'a'; // Compliant
char_t b = '\r'; // Compliant
char_t c = 10; // Noncompliant
char d = 'd'; // Compliant
}
```

In this example, Polyspace flags the use of plain `char` for numeric data.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2015a**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 5-0-12

Signed char and unsigned char type shall only be used for the storage and use of numeric values

## Description

### Rule Definition

*Signed char and unsigned char type shall only be used for the storage and use of numeric values.*

### Rationale

In C/C++, there are three types of `char`:

- Plain `char`
- `signed char`
- `unsigned char`

The signedness of plain `char` is implementation-defined. Plain `char` cannot be interchangeably used with the other types. For instance, you might assume `char` is unsigned and use `unsigned char` to store character. Your implementation might interpret characters as signed. In such a situation, your code might behave in unexpected manner, leading to bugs that are difficult to diagnose.

MISRA C++:2008 limits the use of these three types of `char` for different applications. The `signed` and `unsigned char` type is appropriate for numeric values and storage. The plain `char` is appropriate for character data. Avoid using `signed` or `unsigned char` when you intend to use the plain `char`.

This rule also applies to the different `typedef` of these `char` types, such as `uint8_t` and `int8_t`. See `MISRA C++:2008 Rule 3-9-2`.

### Polyspace Implementation

Polyspace raises a violation of this rule when a plain `char` is implicitly converted to either `signed char` or `unsigned char`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use Plain `char` to Store Characters

```
typedef signed   char         int8_t;
typedef unsigned char         uint8_t;

namespace foo
{
    int8_t        ch_1   =   'a';     // Noncompliant
    uint8_t       ch_2  =   '\r';     // Noncompliant
    char          ch_3  =   'A';      // Compliant
    int8_t        num_1   =   10;     // Compliant
    uint8_t       num_2   =   12U;    // Compliant
    signed char   num_3   =   11;     // Compliant

};
```

In this example, Polyspace flags the use of `signed char` and `unsigned char` to store character data. The character literals are of plain `char` types, and Polyspace flags the implicit conversion of these plain `char` types to explicitly `signed` or `unsigned char` types.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2015a**

## See Also

`Check MISRA C++:2008 (-misra-cpp)` | `MISRA C++:2008 Rule 3-9-2`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-13

The condition of an if-statement and the condition of an iteration- statement shall have type bool

## Description

### Rule Definition

*The condition of an if-statement and the condition of an iteration- statement shall have type bool.*

### Rationale

When you use a non-Boolean expression as a condition for `if`, `while`, and `for` statements, the expression is implicitly converted to `bool`. Such an implicit conversion might make developer intent unclear and hide errors that lead to bugs that are difficult to diagnose. For instance:

```
int flag;
//...
if(flag = 0){
//..
}
```

In the preceding code, it is not clear whether the condition `flag = 0` is intended to be an assignment. The compiler casts the return value of the assignment operation into a `bool`, which is used as the condition for the if statement. If the developer intent is to test whether `flag` equals `0`, then the missing = in the code results in bugs that are difficult to diagnose.

As an exception, conditions of the format `type-specifier-seq declarator` does not need to be Boolean. For instance:

```
while(int* p_int = foo())
```

In this case, the developer intent is clear because of the presence of the type specifier. Avoiding such assignments might make the code difficult to read.

### Polyspace Implementation

Polyspace flags the use of non-Boolean expressions as conditions in `if`, `for`, and `while` statements. As an exception, Polyspace does not flag a non-Boolean condition if the expression is a declaration.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using Non-Boolean Expressions as Conditions

```
typedef unsigned char         uint8_t;
typedef signed   int          int32_t;
typedef          bool         bool_t;
```

```
#include <cstddef>

namespace NS
{
extern int32_t* fn();
extern bool      fn2();
void foo(uint8_t u8)
{
    while (int32_t* p = fn()) {                   // Compliant by exception
        // Code
    }
    // Avoiding assignment altogather in condition statements
    // sometimes lead to clunky code
    do {
        int32_t* p = fn();
        if (NULL == p) {
            break;
        }
        // Code/*...*/
    } while (true);

    while (bool flag = fn2()) {                      // Compliant
        // Code
    }

    if (u8) {}                                       // Non-compliant

}
};
```

In this example, Polyspace flags the use of non-Boolean expressions as conditions.

As the `do-while` loop shows, avoiding declarations in condition statement might lead to code that is difficult to read. Non-Boolean conditions that are declarations are compliant with this rule as exceptions.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-14

The first operand of a conditional-operator shall have type bool

## Description

### Rule Definition

*The first operand of a conditional-operator shall have type bool.*

### Rationale

When you use a non-Boolean expression as the first operand of the ? operator, the expression is implicitly converted to bool. Such an implicit conversion might make developer intent unclear and hide errors that lead to bugs that are difficult to diagnose. For instance:

```
int flag, val;
//...
flag= 1;
val = (flag=0)?2:3;
```

In the preceding code, it is not clear whether the condition flag = 0 is intended to be an assignment. The compiler casts the return value of the assignment operation into a bool, which is used as the condition for the ? operator. In this case, the developer intent might be to test the value of flag. You might expect val to be 3, but it becomes 2 because of the implicit conversion to bool in the first operand.

### Polyspace Implementation

Polyspace raises a violation of this rule if a non-Boolean expression is used as the first argument of the ? operator.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Non-Boolean Expression as the First Argument in ? Operator

```
void foo()
{
    int val, flag;
    flag = 0;
    val = (flag = 0) ? 2 : 3;//Noncompliant
    val = (flag==0)? 2:3; //Compliant
}
```

In this example, Polyspace flags the non-Boolean expression flag = 0 that is used as the first argument of the ? operator.

## Check Information

**Group:** Expressions<br>
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-15

Array indexing shall be the only form of pointer arithmetic

## Description

### Rule Definition

*Array indexing shall be the only form of pointer arithmetic.*

### Rationale

You can traverse an array in two ways:

- Increment or decrement an array index, and then use the array index to access an element.
- Increment or decrement a pointer to the array, and then dereference the pointer.

The first method is clearer and less error-prone. All other forms of explicit pointer arithmetic introduce the risk of accessing unintended memory locations.

As an exception, incrementing or decrementing pointer based iterators is compliant with this rule.

### Polyspace Implementation

The checker flags:

- Arithmetic operations on all pointers, for instance `p+I`, `I+p` and `p-I`, where `p` is a pointer and `I` an integer..
- Array indexing on nonarray pointers.

Polyspace does not flag incrementing or decrementing pointer based iterators, including these standard iterator types:

- `iterator`
- `cont_iterator`
- `reverse_iterator`
- `const_reverse_iterator`

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Explicitly Calculated Pointer Value When Indexing

```
#include<vector>
template < typename IterType >
int sumValues(IterType iter, IterType end)
{
    int result = 0;
    while (iter != end) {
        result += *iter;
        ++iter;  //Noncompliant
    }
    return result;
}
int sumVec(std::vector<int>& v)
{
    int res = 0;
    for (auto it = v.begin(); it != v.end(); ++it) //Compliant by exception
    res += *it;
    return res;
}
int sumVecModern(std::vector<int>& v)
{
    int res = 0;
    for(auto i:v){
        res+=i;
    }
    return res;
}
void foo(int* p_int, int arr_int[])
{
    p_int = p_int + 1; //Noncompliant
    arr_int[0] = arr_int[1];

    p_int[5] = 0; //Noncompliant
    *(p_int + 5) = 0; //Noncompliant
    arr_int[5] = 0.0;

    int a[100];
    std::vector<int> v(100);
    sumValues(&a[0],&a[99]);

}
```

In this example, indexing is done by using array indexing and by calculating the pointer values explicitly. In the function `foo()`:

- Polyspace flags the instances where a pointer value is explicitly calculated, such as `p_int+1` or `*(p_int+5)`.
- Polyspace flags the use of array indexing on the nonarray pointer `p_int`.

Polyspace does not flag uses of array indexing on an array that is compliant with this rule.

Incrementing and decrementing iterators to containers is compliant with this rule by exception. Because `sumVec()` increments an `iterator` object, Polyspace does not flag the increment operation. This exception does not apply to raw pointers. For instance, `sumValues` is instantiated in `foo()` with `int*`. Polyspace flags incrementing the raw pointer. In modern C++, the best practice is to use range-based for loops, as shown in the function `sumVecModern()`.

## Check Information

**Group:** Expressions

**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-16

A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

## Description

### Rule Definition

*A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.*

### Rationale

It is undefined behavior when the result of a pointer arithmetic operation that uses a pointer to an array element does not point to either:

- An element of the array.
- One past the last element of the array. For instance:

  ```
  int arr[3];
  int* res;
  res = arr+3; // res points to one beyond arr
  ```

The rule applies to these operations. `ptr` is a pointer to an array element and `int_exp` is an integer expression.

- `ptr` + `int_exp`
- `int_exp` + `ptr`
- `ptr` - `int_exp`
- `ptr` + +
- ++`ptr`
- --`ptr`
- `ptr`--
- `ptr` [ `int_exp` ]

### Polyspace Implementation

- Single objects that are not part of an array are considered arrays of one element. For instance, in this code example, `arr_one` is equivalent to an array of one element. Polyspace does not flag the increment of pointer `ptr_to_one` because it points to one past the last element of `arr_one`.

  ```
  void f_incr(int* x){
      int* ptr_to_one = x;
      ++ptr_to_one;  // Compliant
  }

  void func(){
      int arr_one=1; // Equivalent to array of one element
      f_incr(&arr_one);
  }
  ```

- Polyspace does not flag the use of pointer parameters in pointer arithmetic operations when those pointers point to arrays. For instance, in this code snippet, the use of `&a1[2]` in `f1` is compliant when you pass an array to `f1`.

  ```
  void f1( int* const a1){
         int* b= &a1[2]; // Compliant
  }
  void f2(){
      int arr[3] {};
      f1(arr);
  }
  ```
- In structures with multiple elements, Polyspace does not flag the result of a pointer arithmetic operation on an element that results in a pointer that points to a different element if the pointer points within the allocated memory of the structure or to one past the last element of the structure.

  For instance, in this code snippet, the assignment to `ptr_to_struct` is compliant because it remains inside `myStruct`, even if it points outside `myStruct.elem1`. Using an index larger than the element dimension to access the content of that element is not compliant, even if the resulting address is within the allocated memory of the structure.

  ```
  void func(){
      struct {
          char elem1[10];
          char elem2[10];
      } myStruct;

      char* ptr_to_struct = &myStruct.elem1[11]; //Compliant
        // Address of myStruct.elem1[11] is inside myStruct
        char val_to_struct = myStruct.elem1[11]; // Non-compliant
  }
  ```
- In multidimensional arrays, Polyspace flags any use of indices that are larger than a subarray dimension to access an element of that subarray. Polyspace does not flag the assignment of the address of that same subarray element if the address is inside the allocated memory of the top-level array.

  For example, in this code snippet, the assignment to pointer `ptr_to_arr` is compliant because the pointer points to an address that is within the allocated memory of `multi_arr`. The assignment to variable `arr_val` is not compliant because the index used to access the subarray element (3) is larger than the dimension of the subarray (2).

  ```
  void func(){
      int multi_arr[5][2];

        // Assigned memory is inside top level array
      int* ptr_to_arr = &multi_arr[2][3]; //Compliant

      // Use of index 3 with subarray of size 2
      int arr_val = multi_arr[2][3]; // Non-compliant
  }
  ```
- Polyspace flags the dereference of a pointer when that pointer points to one past the last element of an array. For instance, in this code snippet, the assignment of `ptr` is compliant, but the dereference of `ptr` is not. `tab+3` is one past the last element of tab.

  ```
  void derefPtr(){
      int tab[3] {};
  ```

```
    int* ptr = tab+3; //Compliant
    int res = *(tab+3); // Non-compliant
}
```

- Polyspace does not raise this checker when the result of a pointer arithmetic operation results in `nullptr`. For instance, consider this code:

  ```
  void g(int *p);

  void add(int* p, int n) {
      g(p + n);   //Compliant}

  void foo() {
      add(nullptr, 0);

  }
  ```

  The pointer arithmetic in `add()` results in a `nullptr`. Polyspace does not flag this operation.

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Pointer Arithmetic by Using Pointers to Array Elements

```
void f_incr(int* x)
{
    int* ptr_to_one = x;
    ++ptr_to_one;  // Compliant
}

void f1(int* const a1)
{
    int* b = &a1[2]; // Compliant
}

int main()
{

    int arr_one = 1; // Equivalent to array of one element
    f_incr(&arr_one);

    int arr[3] {};
    f1(arr);

    struct {
        char elem1[10];
```

```
        char elem2[10];
    } myStruct;

    char* ptr_to_struct = &myStruct.elem1[11]; // Compliant
    ptr_to_struct = &myStruct.elem2[11]; //Non-compliant

    int tab[3] {1, 2, 3};
    int* ptr =  &tab[2];
    int res = tab[2];
    ++ptr; // Compliant
    res = *ptr; //Non-compliant

    return 0;
}
```

In this example:

- The increment of `ptr_to_one` inside `f_incr()` is compliant because the operation results in a pointer that points to one past the last element of array `x`. The integer that is passed to `f_incr()` is equivalent to an array of one element.
- The operation on pointer parameter `a1` inside `f1()` is compliant because the pointer points to array `arr`.
- The first assignment of `ptr_to_struct` is compliant because `elem1[11]` is still inside `myStruct`. The second assignment of `ptr_to_struct` is not compliant because the result of the operation does not point to either inside `myStruct` or to one past the last element of `myStruct`.
- The increment of `ptr` is compliant because the result of the operation points to one past the last element of `tab`. The dereference of `ptr` on the next line is not compliant.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2021a**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-17

Subtraction between pointers shall only be applied to pointers that address elements of the same array

## Description

### Rule Definition

*Subtraction between pointers shall only be applied to pointers that address elements of the same array.*

### Rationale

When you subtract between two pointers to elements in the same array, the result is the distance between the two array elements. If the pointers are null or point to different arrays, a subtraction operation is undefined. If you use the subtraction result as a buffer index, it can cause a buffer overflow.

Before you subtract between pointers to array elements, make sure that they are non-null and that they point to the same array.

### Polyspace Implementation

Polyspace flags subtraction between pointers to elements of different arrays.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Subtracting Pointers to Elements of Different Arrays

```
void foo(){
    int a[10];
    int b[10];
    int distance = a-b;//Noncompliant

}
```

In this example, Polyspace flags the subtraction between `a` and `b`, which are elements of different arrays.

## Check Information

**Group:** Expressions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-18

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array

## Description

### Rule Definition

*>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.*

### Rationale

When you compare two pointers to array elements, the result is the positions of the pointers relative to each other. If the pointers are null or point to different arrays, a comparison operation is undefined.

Before you use >, >=, <, or <= between pointers to array elements, check that they are non-null and that they point to the same array.

### Polyspace Implementation

Polyspace flags the use of >, >=, <, or <= operators between pointers to elements of different arrays.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Comparing Pointers to Elements of Different Arrays

```
bool foo(){
    int a[10];
    int b[10];
    return (a<b); //Noncompliant

}
```

In this example, Polyspace flags the comparison between a and b, which are elements of different arrays.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-0-19

The declaration of objects shall contain no more than two levels of pointer indirection

## Description

### Rule Definition

*The declaration of objects shall contain no more than two levels of pointer indirection.*

### Rationale

If you use pointers with more than two levels of indirection, a developer reading the code might find it difficult to understand the behavior of the code.

### Polyspace Implementation

Polyspace flags all declarations of objects that contain more than two levels of pointer indirection.

- If you use type aliases, the checker includes pointer indirections from the alias in the evaluation of the level of indirection. For instance, in this code snippet, the declaration of `var` is non-compliant. The type of `var` is `const` pointer to a `const` pointer to a pointer to `char`, which is three levels of pointer indirection. The declaration of `var2` has two levels of pointer indirection and is compliant.

  ```
  using ptrToChar = char*;

  void func()
  {
      ptrToChar* const* const var = nullptr; //Non-compliant, 3 levels of indirection
      char* const* const var2 = nullptr; //Compliant, 2 levels of indirection
      //...
  }
  ```
- If you pass an array to a function, the conversion of the array to a pointer to the first element of the array is included in the evaluation of the level of indirection. For instance, in this code snippet, parameter `arrParam` is non-compliant. The type of `arrParam` is a pointer to a pointer to a pointer to `char` (three levels of pointer indirection). The declaration of `arrVar` is compliant because `arrVar` has type array of pointer to pointer to char (two levels of pointer indirection).

  ```
  void func(char** arrParam[])  //Non-compliant
  {
      //...
      char** arrVar[5]; //Compliant
  }
  ```

This checker does not flag the use of objects with more than two levels of indirection. For instance, in this code snippet, the declaration of `var` is non-compliant, but the evaluation of the size of `var` is compliant.

```
#include<iostream>


using charToPtr = char*;

void func()
{
    charToPtr* const* const var = nullptr; //Non-compliant
```

```
    std::cout << sizeof(var) << std::endl; //Compliant

}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 5-0-20

Non-constant operands to a binary bitwise operator shall have the same underlying type

## Description

### Rule Definition

*Non-constant operands to a binary bitwise operator shall have the same underlying type.*

### Rationale

In a binary bitwise operation, if you use operands that have different underlying types, then it is not clear which type you considered when designing the operation. For instance, in this code:

```
uint8_t mask = ~(0x10);
uint16_t value;
value ^= mask; // Non-compliant
```

It is not clear whether you are designing an 8-bit `mask` or a 16-bit `mask`. To avoid such confusion, use the same underlying type for nonconstant operands of binary bitwise operators.

### Polyspace Implementation

If the nonconstant operands of a binary bitwise operator have different underlying types, Polyspace raises a violation of this rule.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using Operands of Mismatched Types in Binary Bitwise Operators

```
#include<cstdint>

void foo()
{
    uint8_t a, b;
    uint16_t c;
    //...
    uint8_t d = a ^ b; //Compliant
    uint16_t e = a ^ c; //Noncompliant

}
```

In this example, Polyspace flags the bitwise operation `a^c` because `a` and `c` are of mismatched underlying data types.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 5-0-21

Bitwise operators shall only be applied to operands of unsigned underlying type

## Description

### Rule Definition

*Bitwise operators shall only be applied to operands of unsigned underlying type.*

### Rationale

Bitwise operations are not meaningful when they are applied to signed operands. Applying bitwise operations to signed operands might result in an unexpected and implementation dependent result. For instance, if you right shift a negative number, the result varies depending on your environment. Avoid using bitwise operations on signed operands.

### Polyspace Implementation

Polyspace raises a violation of this rule when you apply a bitwise operation on a signed operand or expression.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use Bitwise Operations on Unsigned Operands

```
typedef signed   short        int16_t;
typedef unsigned short        uint16_t;

void foo(uint16_t ua, uint16_t ub, int16_t a)
{
    if   (   ( ua & a ) == 0x1234U ) {}        // Noncompliant
    if   (   ( ua | ub ) == 0x1234U ) {}       // Compliant
    if   (   ~a == 0x1234U ) {}                // Noncompliant
    if   (   ~ua == 0x1234U ) {}               // Compliant
}
```

In this example, Polyspace flags the use of bitwise operations on signed operand `a`. Bitwise operations that involve unsigned operands such as `ua`, `ub`, and the hexadecimal number `0x1234U`, are compliant with this rule.

## Check Information

**Group:** Expressions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-2-1

Each operand of a logical && or || shall be a postfix-expression

## Description

### Rule Definition

*Each operand of a logical && or || shall be a postfix-expression.*

### Rationale

This rule effectively requires that operands of a logical && or `||` operation be appropriately parenthesized. For instance, instead of `a + b || c`, the rule requires `(a + b) || c` or `a + (b || c)`. In both compliant cases, the left operand of `||`, that is `(a + b)` or `b`, is a primary expression and therefore also a postfix expression. For more information on postfix expressions, see the C++03 Standard (Section 5.2).

Enclosing operands in parentheses improves readability of code and makes sure that the operations occur in the order that the developer intends.

### Polyspace Implementation

The checker raises a violation if a logical && or `||` operand is not a postfix expression.

A postfix expression can be a primary expression such as a simple identifier or a combination of identifiers enclosed in parentheses, but also one of the following:

- Function call such as `func()`.
- Array element access such as `arr[]`.
- Structure member access such as `aStructVar.aMember`.

For the complete list of postfix expressions, see the C++03 Standard (Section 5.2).

The checker allows exceptions on associative chains such as `(a && b && c)` or `(a || b || c)`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant and Compliant Expressions Involving Logical Operations

```
bool Operations(bool a, bool b, bool c, bool priority) {
    bool res;
    if(priority) {
        res = a && b || c;  //Noncompliant
    }
    else {
        res = a && (b || c); //Compliant
    }
    return res;
}
```

In this example, the expression `a && b || c` violates the rule because the right operand of && and the left operand of `||` are not postfix expressions.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 5-2-2

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of `dynamic_cast`

## Description

### Rule Definition

*A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of `dynamic_cast`.*

### Rationale

A `virtual` base implies that multiple classes might be derived from it.

```
class Base{};
class childA: virtual public Base{};
class childB: virtual public Base{};
class childFinal: public childA, public childB{};;
```

In the preceding code, the derived classes `childA` and `childB` share the same instance of the class `Base`. The necessary pointer arithmetic is unknown at compile time when you cast from `Base*` to `childA*` or `childB*`. The offset of the child classes compared to `Base` is known only at run time.

Use `dynamic_cast` when downcasting a pointer to a virtual base class into a pointer to a child class. Using any other casting operation for this purpose results in an undefined behavior.

### Polyspace Implementation

Polyspace raises a violation if these conditions are met:

- A virtual base class is downcast.
- The casting operation is not done by using `dynamic_cast`

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use `dynamic_cast` to Downcast Pointers to `virtual` Base

```
class Base{};
class childA: virtual public Base{};
class childB: virtual public Base{};
class childFinal: public childA, public childB{};

void bar(){
    Base* pB;
    //...
    static_cast<childA*>(pB);//Noncompliant
}
```

In this example, the pointer to a `virtual` base `pB` is downcast to a pointer to a derived class. The information needed to cast `pB` to a `childA*` type is known at run time only. Using `static_cast` to perform these casts results in an undefined behavior. Polyspace raises a violation. This issue might also be reported as a compilation error.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 5-2-3

Casts from a base class to a derived class should not be performed on polymorphic types

## Description

### Rule Definition

*Casts from a base class to a derived class should not be performed on polymorphic types.*

### Rationale

Explicitly casting a polymorphic base class into a derived class bypasses the layers of abstraction in the implementation hierarchy, resulting in increased coupling and dependency between the classes.

When using a polymorphic class, avoid performing explicit downcasts. A pointer to the polymorphic base class does not require explicit casting into a pointer to a derived class.

### Polyspace Implementation

Polyspace raises a violation if a base class is cast to a derived class.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Downcasting In a Polymorphic Class Hierarchy

```
class Base{ //Abstract base class
    //...
public:
    virtual bool getStatus() const = 0;
};
class Derived: public Base{
    //...
public:
    virtual bool getStatus() const{ //Implementation of getStatus()
        //...
    }
};
void printStatus(const Base& B){
    const Derived* D = dynamic_cast<const Derived*>(&B);//Noncompliant
    D->getStatus();
}
void printStatus_compliant(const Base& B){
    B.getStatus();//Compliant
}
```

In this example, the `Base` class pointer `B` is cast to a `Derived` class pointer in `printStatus`. This cast is noncompliant and Polyspace raises a violation. The function `getStatus_compliant()` shows a compliant implementation of this functionality, where `virtual` functions invoke the `Derived` class member function through a pointer to the `Base` class.

### Check Information

**Group:** Expressions
**Category:** Advisory

## Version History

**Introduced in R2013b**

### See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-2-4

C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used

## Description

### Rule Definition

*C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.*

### Rationale

C-style casts are difficult to distinguish in the source code. The developer intent in casting from one type to another is not communicated. Code that contains C-style casts are difficult to understand and debug. Avoid C-style casts.

C++ introduces explicit casting operations that are easily identified and that clearly communicate the developer intent behind each cast. Use these casting operations instead.

### Polyspace Implementation

Polyspace flags c-style casts in your code. Compliant C++ style casting operations include:

- std::static_cast
- std::reinterpret_cast
- std::const_cast
- std::dynamic_cast
- Cast by using {} notation

In addition, the reference_cast operation from the boost library and the safe_cast operation from the Miscrosoft library are also allowed.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid C Style Casts

```
#include <cstdint>
class Base
{
public:
    explicit Base(std::int32_t) {}
    virtual void Fn() noexcept {}
};
class Derived : public Base
```

```
{
public:
    void Fn() noexcept override {}
};
class myClass
{
};
std::int32_t G() noexcept
{
    return 7;
}

void Foo() noexcept(false)
{
    Base obj = Base{10};        // Compliant
    const Base obj3(5);
    Base* obj4 = const_cast<Base*>(&obj3);        //Compliant
    Base* obj2;
    myClass* d1 = reinterpret_cast<myClass*>(obj4); // Compliant
    Derived* d2 = dynamic_cast<Derived*>(obj2);       // Compliant
    std::int16_t var1 = 20;
    std::int32_t var2 = static_cast<std::int32_t>(var1);      // Compliant
    std::int32_t var4 = 10;
    float f1 = static_cast<float>(var4);                    // Compliant

    std::int32_t var5 = static_cast<std::int32_t>(f1);      //Compliant

    std::uint32_t var7 = std::uint32_t{0};//Compliant
    static_cast<void>(G());                               //Compliant
}
void Bar(){
    Base obj = Base{1};
    Base* obj2 = (Base*)(&obj); //Noncompliant
    std::int16_t var1 = 20;
    std::int32_t var3 = (std::int32_t)var1;                   // Noncompliant
    float f = (float)var3;                                 // Noncompliant
    std::int32_t var6 = (std::int32_t)f;                   // Noncompliant
}
```

In this example, the function `Foo()` shows several C++ style casting operations. These operations are easily identified, and their purpose is clear. For instance, in `Base* obj4 = const_cast<Base*>(&obj3)`, it is clear that the cast operation removes the const qualifier from `obj3`. These cast operations are compliant and Polyspace does not flag them.

The function `Bar()` shows several C style casts. These operations are difficult to find understand. Polyspaceflags these C-style casts.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-2-5

A cast shall not remove any const or volatile qualification from the type of a pointer or reference

## Description

### Rule Definition

*A cast shall not remove any const or volatile qualification from the type of a pointer or reference.*

### Rationale

Removing the `const` or `volatile` qualification from a pointer or reference might be unexpected. Consider this code:

```
void foo(const char* p){
  *const_cast< char_t * >( p ) = '\0'
}
```

The function `foo()` accepts a `const` pointer to a `char`. The caller of this functions expects that the parameter `p` remains unchanged. Modifying `p` in `foo()` by converting it to a non-`const` pointer is unexpected. If `*p` dereferences to a `const` character, this modification might lead to unexpected behavior. Avoid casting the `const` or `volatile` away from a pointer or reference.

### Polyspace Implementation

Polyspace raises a violation of this rule if you remove the `const` or `volatile` qualification from the type of a pointer or a reference by using a casting operation.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Casting Away `Const` from Pointers

```
void foo(const char* p){
  *const_cast< char * >( p ) = '\0';//Noncompliant
}

void foo1(volatile char* p){
  (char*) p ;//Noncompliant
}
```

In this example, Polyspace flags the casting operations that cast away the `const` and `volatile` qualifiers from pointers.

## Check Information

**Group:** Expressions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-2-6

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type

## Description

### Rule Definition

*A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.*

### Rationale

Because calling a function through a pointer with incompatible type results in undefined behavior, the rule forbids these cast operations:

- Casting from a function pointer to any other type.
- Casting from a function pointer to another function pointer, if the function pointers have different arguments and return types.

### Polyspace Implementation

Polyspace raises a violation of this rule if the source type of a cast operation is a pointer to a function.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Casting Function Pointers

```
void* (*func1) (int);
void* (*func2)(float);
void bar(void* (*)(int));
void foo(){

    bar(reinterpret_cast<void* (*) (int)> (func2)); //Noncompliant
    bar(func1);
}
```

In this example, the function `bar` accepts a pointer to a function that takes an `int` as an input and returns a `void*` pointer. In `foo()`, the function pointer `func2` is converted so that it can be passed to `bar`. Polyspace flags the noncompliant conversion of function pointers.

## Check Information

**Group:** Expressions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-2-7

An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly

## Description

### Rule Definition

*An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.*

### Rationale

If you convert a pointer to a pointer of unrelated type, the result of the operation is unspecified. To avoid unexpected results, do not convert a pointers to an unrelated pointer type.

### Polyspace Implementation

The checker flags all pointer conversions including between a pointer to a `struct` object and a pointer to the first member of the same `struct` type.

Indirect conversions from a pointer to non-pointer type are not detected.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 5-2-8

An object with integer type or pointer to void type shall not be converted to an object with pointer type

## Description

### Rule Definition

*An object with integer type or pointer to void type shall not be converted to an object with pointer type.*

### Polyspace Implementation

The checker allows an exception on zero constants, such as `0x0`, `0`, or `0U`.

Objects with pointer type include objects with pointer-to-function type.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Converting to Pointers

In this example, Polyspace flags the conversion of integers to pointers. Polyspace does not flag converting zero constants such as `0`, `0x0`, or `0U` to pointers. Such conversions are equivalent to zero initialization of a pointer, which has a specified behavior.

```
void foo ()
{
  void *p1;
  void *p2;
  void *p3;

  p1 = static_cast < void *>(0x0);    //Compliant
  p2 = static_cast < void *>(0U);    //Compliant
  p3 = static_cast < void *>(0);    //Compliant

  p1 = (void *) 0xDEADBEEF;    //Noncompliant
  p2 = (void *) 03;        //Noncompliant
  p3 = (void *) 12345678U;    //Noncompliant
}
```

## Check Information

**Group:** Expressions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-2-9

A cast should not convert a pointer type to an integral type

## Description

### Rule Definition

*A cast should not convert a pointer type to an integral type.*

### Rationale

The C++ standard specifies only the minimum size required for integral types. The implemented size of integral types and pointers depends on your hardware and development environment. In an environment where pointers have a larger size than integral types, casting a pointer type to an integral type results in an overflow. Avoid casting pointers to integral types.

### Polyspace Implementation

Polyspace flags a casting operation if it converts a pointer type variable to an integral type variable.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Casting Pointers to Integers

```
void foo(){
    int* pInt;
    //...
    int address = reinterpret_cast<int>(pInt);//Noncompliant
}
```

In this example, the pointer `pInt` is cast to an int. Depending on your environment, this operation might result in an overflow. For instance, in a 64-bit system, the size of `pInt` might be 64-bit and the size of `address` might be 32-bit. Polyspace flags this casting operation.

## Check Information

**Group:** Expressions
**Category:** Advisory

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-2-10

The increment ( ++ ) and decrement ( -- ) operators should not be mixed with other operators in an expression

## Description

### Rule Definition

*The increment ( ++ ) and decrement ( -- ) operators should not be mixed with other operators in an expression.*

### Rationale

Using the increment and decrement operators with other operators in an expressions results in code that is difficult to read. Such code might lead to undefined behavior.

### Polyspace Implementation

Polyspace flags an expression if it contains the increment or decrement operators mixed with other operators. If an expression contains multiple increment or decrement operators mixed with other operators, Polyspace flags the first increment or decrement operator.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Mixing Increment or Decrement Operators with Other Operators

```
void foo(int a, int b){
    int c = ++a + b--; //Noncompliant
    if(--c + --a - ++b){ //Noncompliant
        //...
    }
}
```

Polyspace flags the mixing of the ++ and `--` operators with other mathematical operators in an expression.

## Check Information

**Group:** Expressions
**Category:** Advisory

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-2-11

The comma operator, && operator and the || operator shall not be overloaded

## Description

### Rule Definition

*The comma operator, && operator and the || operator shall not be overloaded.*

### Rationale

When you overload an operator, the overloaded operator behaves as a function call. The comma operator, the && operator, and the `||` operator have certain behaviors that cannot be replicated by their overloaded counterpart. For instance, a compiler might short circuit the builtin && or `||` operators. But such short circuiting is not possible when you use an overloaded version of these operators.

Overloading these operators creates confusion about how these operators behave. Avoid overloading the comma operator, the && operator, and the `||` operator.

### Polyspace Implementation

Polyspace flags the overloading of these operators:

- Comma operator
- && operator
- `||` operator

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Do Not Overload the && Operator

```
class flag{/**/};
class Util
{
public:
flag getValue ( );
flag setValue ( int const & );
};

bool operator && ( flag const &, flag const & ); // Noncompliant
void f2 ( Util & in3, Util & in4 )
{
in3.getValue ( ) && in4.setValue ( 0 ); // Both operands evaluated
}
```

In this example, the && operator is overloaded for the class `flag`. In `f2()`, the overloaded operator is used. The overloading prevent the short circuiting. The behavior of the overloaded operator might be unexpected. Polyspace flags the overloading of the && operator.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-2-12

An identifier with array type passed as a function argument shall not decay to a pointer

## Description

### Rule Definition

*An identifier with array type passed as a function argument shall not decay to a pointer.*

### Rationale

When you pass an array to a function as a pointer, the size information of the array becomes lost. Losing information about the array size might lead to confusion and unexpected behavior.

Avoid passing arrays as pointers to function. To pass arrays into a function, encapsulate the array into a class object and pass the object to functions. Starting in C++11, the standard template library implements several container classes that can be used to pass an array to a function. C++20 has the class `std::span`, which preserves the size information.

### Polyspace Implementation

Polyspace raises a violation when you use an array in a function interface.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using Arrays in Function Interfaces

```
void f1( int p[ 10 ] ); // Noncompliant

void f2( int ( &p )[ 10 ] );// Compliant
void foo ()
{
int a[ 10 ];
f1( a );

f2( a );
}
```

In this example,the interface of `f1()` uses an array. When you pass the array `a[10]` to `f1()` as a pointer, the size of the array `a` is lost. Polyspace flags the declaration of `f1()`. If you pass arrays to function while preserving the dimensionality information, as shown by `f2()`, Polyspace does not raise a violation.

## Check Information

**Group:** Expressions

**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-3-1

Each operand of the ! operator, the logical && or the logical || operators shall have type bool

## Description

### Rule Definition

*Each operand of the ! operator, the logical && or the logical || operators shall have type bool.*

### Rationale

Use of non-`bool` operands with the logical operators `!`, && or `||`, is likely to indicate a programming error.

### Polyspace Implementation

The checker reports a violation if the operands of the logical operators `!`, && or `||`, are not effectively boolean. Allowed operands are:

- Variables of type `bool` or specified as effectively boolean using the option `Effective boolean types (-boolean-types)`.
- Expressions that compare two variables using operators such as == or < and return a boolean.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Logical && With Boolean and Non-Boolean Operands

In this example, both operands of the logical && in the `if` condition are effectively boolean and comply with the rule. The right operand of the logical && in the `else if` condition has a non-boolean type (`int32_t`) and violates the rule.

```
#include <cstdint>

class Pair
{
    int32_t leftVal;
    int32_t rightVal;
public:
    bool operator==(const Pair& anotherPair) {
        return(((*this).leftVal == anotherPair.leftVal)
             &&((*this).rightVal == anotherPair.rightVal));
    }
};

Pair getAPair();
int32_t checkPair(const Pair*);
```

```
void main() {
    Pair aPair = getAPair();
    Pair anotherPair = getAPair();
    Pair athirdPair = getAPair();

    if((aPair == anotherPair) &&
       (aPair == athirdPair)) //Compliant
       {}
    else if((aPair == anotherPair) && //Noncompliant
            (checkPair(&athirdPair)))
       {}
}
```

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-3-2

The unary minus operator shall not be applied to an expression whose underlying type is unsigned

## Description

### Rule Definition

*The unary minus operator shall not be applied to an expression whose underlying type is unsigned.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 5-3-3

The unary & operator shall not be overloaded

## Description

### Rule Definition

*The unary & operator shall not be overloaded.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 5-3-4

Evaluation of the operand to the sizeof operator shall not contain side effects

## Description

### Rule Definition

*Evaluation of the operand to the sizeof operator shall not contain side effects.*

### Polyspace Implementation

The checker does not show a warning on volatile accesses and function calls

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 5-8-1

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand

## Description

### Rule Definition

*The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 5-14-1

The right hand operand of a logical && or || operator shall not contain side effects

## Description

### Rule Definition

*The right hand operand of a logical && or || operator shall not contain side effects.*

### Rationale

When evaluated, an expression with side effect modifies at least one of the variables in the expression. For instance, `n++` is an expression with side effect.

The right-hand operand of a:

- Logical && operator is evaluated only if the left-hand operand evaluates to true.
- Logical `||` operator is evaluated only if the left-hand operand evaluates to false.

In other cases, the right-hand operands are not evaluated, so side effects of the expression do not take place. If your program relies on the side effects, you might see unexpected results in those cases.

### Polyspace Implementation

The checker flags logical && or `||` operators whose right operands are expressions that have side effects. Polyspace assumes:

- Expressions that modifies at least one of its variables have side effects.
- Explicit constructors or conversion functions that are declared but not defined have no side effects. Defined conversion functions have side effects.
- Volatile accesses and function calls have no side effects.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Side Effects in Right Operand of Logical Operation

```
class real32_T {
public:
    real32_T() = default;

    /* Casting operations */
    explicit real32_T(float a) {
        // ...
    }
    /* Relational operators */
    bool operator==(real32_T a) const;
    bool operator>(real32_T a) const;
};

void bar() {
    real32_T d;



    if ((d == static_cast<real32_T>(0.0F))
    || (static_cast<real32_T>(0.0F) > d)) {//Noncompliant
        /**/
    }
}



void foo(int i, int j){
    if(i==0 && ++j==i){ //Noncompliant
        --i;
    }
}
```

In the function `foo`, the right operand of the && operator contains an increment operation, which has a side effect. Polyspace flags the operator. In the function `bar`, the right operand of the `||` operator contains a conversion function that is implemented in the class. Polyspace considers such constructor to have side effects. Because the right operator has side effects, the operator is flagged.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-18-1

The comma operator shall not be used

## Description

### Rule Definition

*The comma operator shall not be used.*

### Rationale

The comma operator takes two operands. It evaluates the operators from left to right, discards the value of the left operand,and returns the value of the right operand. This operator has the lowest operator precedence. These properties make the use of the comma operator nonintuitive. For instance, consider this code:

```
array[i++,j] = 1
```

At a glance, it might appear that the preceding code accesses a multidimensional array. In fact, this code is equivalent to:

```
i++;
array[j] = 1;
```

The use of the comma operator makes the code difficult to read and maintain. To avoid confusion, do not use the comma operator. Refactor the expression into multiple statements instead.

### Polyspace Implementation

Polyspace flags the use of comma operator. Violations are not raised when you use comma operator for function calls and initializations.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Comma Usage in C++ Code

```
typedef signed int abc, xyz, jkl;
static void func1 ( abc, xyz, jkl );       /* Compliant - case 1 */
int foo(void)
{
    volatile int rd = 1;                   /* Compliant - case 2*/
    int var=0, foo=0, k=0, n=2, p, t[10];  /* Compliant - case 3*/
    int abc = 0, xyz = abc + 1;            /* Compliant - case 4*/
    int jkl = ( abc + xyz, abc + xyz );    /* Noncompliant - case 1*/
    var = 1, foo += var, n = 3;          /* Noncompliant - case 2*/
    var = (n = 1, foo = 2);              /* Noncompliant - case 3*/
    for ( int *ptr = &t[ 0 ],var = 0 ;
          var < n; ++var, ++ptr){}    /* Noncompliant - case 4*/
    if ((abc,xyz)<0) { return 1; }         /* Noncompliant - case 5*/
}
```

In this example, the code shows various uses of commas in C code.

**Noncompliant Cases**

| Case | Reason for noncompliance |
|---|---|
| 1 | When reading the code, it is not immediately obvious what `jkl` is initialized to. For example, you could infer that `jkl` has a value `abc+xyz`, `(abc+xyz)*(abc+xyz)`, `f((abc+xyz),(abc+xyz))`, and so on. |
| 2 | When reading the code, it is not immediately obvious whether `foo` has a value 0 or 1 after the statement. |
| 3 | When reading the code, it is not immediately obvious what value is assigned to `var`. |
| 4 | When reading the code, it is not immediately obvious which values control the `for` loop. |
| 5 | When reading the code, it is not immediately obvious whether the `if` statement depends on `abc`, `xyz`, or both. |

**Compliant Cases**

| Case | Reason for compliance |
|---|---|
| 1 | Using commas to call functions with variables is allowed. |
| 2 | Comma operator is not used. |
| 3 & 4 | When using the comma for initialization, the variables and their values are immediately obvious. |

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 5-19-1

Evaluation of constant unsigned integer expressions should not lead to wrap-around

## Description

### Rule Definition

*Evaluation of constant unsigned integer expressions should not lead to wrap-around.*

### Rationale

Unsigned integer expressions do not strictly overflow, but instead wraparound. Although there may be good reasons to use modulo arithmetic at run time, intentional use at compile time is less likely and might imply logic errors.

### Polyspace Implementation

Polyspace flags the constant expressions that might wraparound.

Different compilers might define compile-time constants differently. In the following code, `c+1u` is considered a constant expression by GCC compilers, but not by the standard C++ compiler.

```
const uint16_t c = 0xffffu;
uint16_t y = c + 1u;
```

Whether you see a violation of this rule in the preceding code might depend on your compiler.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Unsigned Integer Wraparounds in Constant Expression

```
#define BEGIN        0x8000
#define FIN          0xFFFF
#define POS          0x8000
#if ( ( BEGIN + POS ) > FIN )
//....
#endif
#if ( ( ( FIN - BEGIN ) - POS ) < 0 )   //Noncompliant
//...
#endif

  void fn ( )
  {
    if ( ( BEGIN + POS ) > FIN ) {   // Noncompliant
      //...
    }
  }
```

In this example, the constant expressions `(( FIN - BEGIN ) - POS)` and `(( BEGIN + POS ) > FIN)` might lead to wraparounds. Polyspace flag these expressions.

## Check Information

**Group:** Expressions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-2-1

Assignment operators shall not be used in sub-expressions

## Description

### Rule Definition

*Assignment operators shall not be used in sub-expressions.*

### Rationale

When used in a subexpression, assignment operators have side effects that are difficult to predict. These side effects might produce results contrary to developer expectations. This rule helps in avoiding confusion between the assignment operator (=) and the equal to operator (==). Do not use assignment operators in subexpressions.

### Polyspace Implementation

Polyspace raises this defect whenever a subexpression contains an assignment operator.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Assignment Operators in Sub-Expressions Are Noncompliant

```
#include <cstdint>

bool example(int x, int y)
{
    if (x == 10)         //Compliant
    {
        return true;
    }

       if ((x = y) == 0)            //Noncompliant
    {
        return false;
    }

    return false;
}
```

Because the assignment operator = is used in the subexpression `(x = y)`, Polyspace flags it as noncompliant.

## Check Information

**Group:** Statements

**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-2-2

Floating-point expressions shall not be directly or indirectly tested for equality or inequality

## Description

### Rule Definition

*Floating-point expressions shall not be directly or indirectly tested for equality or inequality.*

### Rationale

Due to the inherent rounding errors of floating-point numbers, there is no way to reliably compare floating-point numbers for equality. This includes both direct and indirect tests of equivalency. Comparisons of floating-point numbers can result in a false outcome when you expect equivalence. This behavior is unpredictable and can vary between implementations.

Avoid using floating-point numbers for equivalence comparisons. Alternatively, write a library that implements your comparison operations. Take into account the magnitude of numbers to compare as well as floating-point granularity during creation of this library.

### Polyspace Implementation

The rule checker detects the use of == or != with floating-point variables or expressions. Additionally, the rule checker detects indirect tests of equality or inequality of floating-point variables. For example:

```
float x, y = 0.0;
if ((x < y) || (x > y))           //Noncompliant
{
    //...
}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Floating-Point Number Comparisons of Equivalence

This code contains different examples of floating-point number comparisons. Polyspace reports a violation for each of these floating-point tests of equality or inequality.

- `x == y`, due to the use of ==
- `x == 0.0f`, due to the use of `!=`
- `((x <= y) && (x >= y))`, due to an indirect comparison of equality
- `((x < y) || (x > y))`, due to an indirect comparison of inequality

```
#include <cmath>
#include <limits>
```

```
void main()
{
    float x, y = 0.0;
    if (x == y)          //Noncompliant
    {
        //...
    }
    if (x != 0.0f)       //Noncompliant
    {
        //...
    }
    if ((x <= y) && (x >= y))         //Noncompliant
    {
        //...
    }
    if ((x < y) || (x > y))           //Noncompliant
    {
        //...
    }
}
```

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-2-3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character

## Description

### Rule Definition

*Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.*

### Rationale

A null statement:

```
;
```

used on a line with other code can indicate a programming error. If you want to enter a null statement, place the statement on a line by itself.

A comment can follow a null statement, but use a white space character between the null statement and the comment to provide a visual cue for reviewers.

### Polyspace Implementation

The rule checker considers a null statement as a line where the first character excluding comments is a semicolon. The rule checker reports violations for situations where:

- Comments appear before the null statement.

  For instance:

  ```
  /* wait for pin */ ;
  ```
- Comments appear immediately after the semicolon without a white space in between.

  For instance:

  ```
  ;// wait for pin
  ```

The rule checker also reports a violation when a second statement appears on the same line following the null statement.

For instance:

```
; count++;
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Proper Formatting for Null Statement

In this example, there are three different versions of comments after a null statement.

- The first option is compliant. The comment comes after the null statement, the null statement is the only statement on the line, and there is a white space separating the semicolon and the comment.
- The second option is noncompliant. The comment comes before the null statement.
- The third option is noncompliant. A white space does not separate the semicolon and comment.

```
#include <iostream>

void main()
{
    std::cout << "Hello World" << std::endl;
     ; // comment here - Compliant
     /* comment here */ ; // Noncompliant
     ;//comment here - Noncompliant

}
```

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

Check MISRA C++:2008 (-misra-cpp)

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-3-1

The statement forming the body of a switch, while, do while or for statement shall be a compound statement

## Description

### Rule Definition

*The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.*

### Rationale

A compound statement is included in braces.

If a block of code associated with an iteration or selection statement is not contained in braces, you can make mistakes about the association. For example:

- You can wrongly associate a line of code with an iteration or selection statement because of its indentation.
- You can accidentally place a semicolon following the iteration or selection statement. Because of the semicolon, the line following the statement is no longer associated with the statement even though you intended otherwise.

This checker enforces the practice of adding braces following a selection or iteration statement even for a single line in the body. Later, when more lines are added, the developer adding them does not need to note the absence of braces and include them.

### Polyspace Implementation

The checker flags `for` loops where the first token following a `for` statement is not a left brace, for instance:

```
for (i=init_val; i > 0; i--)
   if (arr[i] < 0)
      arr[i] = 0;
```

Similar checks are performed for `switch`, `for` and `do..while` statements.

The second line of the message on the **Result Details** pane indicates which statement is violating the rule. For instance, in the preceding example, the second line of the message states that the `for` loop is violating the rule.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Statements

**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-4-1

An if ( condition ) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement

## Description

### Rule Definition

*An if ( condition ) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.*

### Rationale

If you use single statements as bodies of if or if...else constructs, then attempting to change the body into a compound statement might result in logic errors and unexpected results. This rule ensures that braces are not missed or forgotten when attempting to turn a single statement body into a compound statement body. Failure to use compound statements might provide unexpected results and cause developer confusion. Use `{ }` braces to create compound statements. Use braces even when the if statement is simple and contains only a single statement.

### Polyspace Implementation

Polyspace raises this defect whenever a compound statement does not immediately follow an if statement, else-if statement, or else statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Using Compound Statements in If...Else Conditionals

```
#include <cstdint>

int example(int test, int result)
{
    if (test > 5)
    {
        test--;
        result = test + result;   //Compliant
    }
    else if (test <= 5)
    {
        result = test - result;   //Compliant
    }
    else                          //Noncompliant
        result = test;

    return result;
}
```

Because the else statement does not use `{ }` braces to form a compound statement, Polyspace flags it as noncompliant.

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 6-4-2

All if … else if constructs shall be terminated with an else clause

## Description

### Rule Definition

*All if … else if constructs shall be terminated with an else clause.*

### Rationale

Ending an if... else if construct with an `else` statement is defensive programming. This final `else` statement acts as a fail-safe in case a unique situation occurs where the code progresses past the `if` and `else if` statements.

When an `if` statement is followed by one or more `else if` statements, follow the final `else if` statement with an `else` statement. Within the `else` statement provide an action. If no action is needed, provide a comment as to why no action is taken.

### Polyspace Implementation

Polyspace raises this defect whenever an `if … else if` construct does not end with an `else` statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### If Construct With No Else If Statements

```
#include <cstdint>

int example(int test, int result)
{
    if (test > 5)
    {
        test--;
        result = test + result;
    }

    return result;
}
```

Because no `else if` statement exists in this `if` construct, no final `else` statement is required within the construct.

### If Construct Containing Else-If Statements

```
#include <cstdint>
```

```
int example(int test, int result)
{
    if (test > 5)
    {
        test--;
        result = test + result;
    }
    else if (test <= 5)             //Noncompliant
    {
        result = test - result;
    }

    return result;
}
```

Because the final `else if` statement is not followed by a closing else statement, Polyspace marks it as noncompliant. Even though there should be no situation where a value of test progresses past both the `if` and `else if` statements, the additional else statement is required.

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-4-3

A switch statement shall be a well-formed switch statement

## Description

### Rule Definition

*A switch statement shall be a well-formed switch statement.*

### Rationale

In addition to the C++ standard syntax rules, MISRA defines their own syntax rules for creating well-formed switch statements. These additional syntax rules create a consistent structure for switch statements.

The additional MISRA syntax rules include:

| Rule | Syntax |
|---|---|
| switch-statement | `switch` (condition) {case-label-clause-list default-label-clause$_{opt}$} |
| case-label-clause-list | case-label case-clause$_{opt}$<br>case-label-clause-list case-label case-clause$_{opt}$ |
| case-label | `case` const-expression |
| case-clause | case-block-seq$_{opt}$ `break` ;<br>case-block-seq$_{opt}$ `throw` assignment-expression$_{opt}$ ;<br>{ statement-seq$_{opt}$ `break` ; }<br>{ statement-seq$_{opt}$ `throw` assignment-expression$_{opt}$ ; } |
| default-label-clause | default-label default-clause |
| default-label default-clause | `default` : case-clause |
| case-block | expression_statement<br>compound_statement<br>selection_statement<br>iteration_statement<br>try_block |
| case-block-seq | case-block<br>case-block-seq case-block |

These terms from the table are defined as such:

- switch-label — Either a case-label or default-label.
- case-clause — The code between any two switch-labels.
- default-clause — The code between the default-label and the end of the switch statement.
- switch-clause — Either a case-clause or a default-clause.

The MISRA C++ switch syntax rules do not include the following statements, but do permit them within the compound statements that form the body of a switch-clause statement.

- labelled_statement
- jump_statement
- declaration_statement

### Polyspace Implementation

The rule checker reports a violation in these situations:

- A statement occurs between the switch statement and the first case statement.

  For instance:

  ```
  switch(ch) {
    int temp;
    case 1:
      break;
    default:
      break;
  }
  ```
- A label or a jump statement such as goto or return occurs in the switch block.
- A variable is declared in a case statement that is outside any block.

  For instance:

  ```
  switch(ch) {
    case 1:
      int temp;
      break;
    default:
      break;
  }
  ```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Adhere to MISRA Rules for Well-Formed Switch Statements

The example fails to follow two of the well-formed switch statement rules from MISRA.

- The statement int temp; occurs between the switch statement and the first case statement.

- `case 2` contains a return statement.

Polyspace reports a single violation for the entire `switch` statement despite multiple violations within the `switch` statement.

```
int example(int x)
{
    switch (x) {            //Noncompliant
        int temp;
        case 0:
            break;
        case 1:
        case 2:
            return x;
        default:
            break;
    }

    return x;
}
```

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-4-4

A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

## Description

### Rule Definition

*A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.*

### Rationale

Placing a case-label or default-label of a switch statement in different scopes might result in unstructured code. Unstructured code might lead to unexpected behavior resulting in developer confusion. To prevent this issue, all case-labels and the default-label must be at the same scope of the compound statement forming the body of the switch statement.

### Polyspace Implementation

Polyspace raises this defect whenever a case-label belongs to any scope other than the switch statement body.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Case-Label Nested Within Another Case-Label Is Noncompliant

```
#include <cstdint>

int y, sum;

int heresASum(int a, int b)
{
    sum = a + b;
    return sum;
}

int example(int x)
{
    switch (x) {
    case 1:                         //compliant
        if (y > 0) {
        case 2:                     //noncompliant
            y = heresASum(x, y);
            break;
        }
        break;
    case 3:
        break;

    default:
        break;
    }

    return x;
}
```

Because the case-label `case 2` in nested under the `case 1` case-label, it is considered in a different scope from the other case-label statements and the default-label statement. All of these other case-label statements are in the same scope of the body of the switch-label statement.

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-4-5

An unconditional throw or break statement shall terminate every non - empty switch-clause

## Description

### Rule Definition

*An unconditional throw or break statement shall terminate every non - empty switch-clause.*

### Rationale

If a throw or break statement is not used at the end of a switch-clause, control flow falls into the next switch-clause. If unintentional, this behavior might cause unexpected results. Using a throw or break statement helps to prevent unintentional fall-through behavior. Use a throw or break statement as the last statement of each case-clause and the default-clause.

Using an empty case-label is acceptable when utilizing fall-through to group together multiple clauses that otherwise require identical statements.

### Polyspace Implementation

Polyspace raises this defect whenever a case-label contains any statements and a throw or break statement is not the final statement of the case-label.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Noncompliant Unintentional Fall Through Case-Label

```
#include <cstdint>

int x, y = 2;

int example(int x)
{
    switch (x) { //Noncompliant error shows here
    case 0:              //Compliant empty fall through
    case 1:
        break;           //Compliant break
    case 2:
        x = y ^ 2;       //Unintentional fall through
    case 3:              //Compliant throw
        throw;

    default:             //Compliant break
        break;
    }
    return x;
}
```

Because `case 2` does not contain a throw or break statement, it falls over into `case 3`. This type of fall through is noncompliant.

Because it is an empty case-label, `Case 0` will fall through to `case 1`. This is a compliant empty case-label fall through.

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-4-6

The final clause of a `switch` statement shall be the default-clause

## Description

### Rule Definition

*The final clause of a `switch` statement shall be the default-clause.*

### Rationale

It is a good defensive programming technique to always use a `default` clause as the final clause of a `switch` statement. The `default` clause must either take appropriate action or contain a comment as to why the `default` clause takes no action.

### Polyspace Implementation

The rule checker detects `switch` statements that do not have a final `default` clause.

The rule checker does not report a violation if the `switch` variable is an enumeration with finite number of values and you have a `case` clause for each value. For instance:

```
enum Colors { RED, BLUE, YELLOW } color;

switch (color) {
      case RED:
        break;
      case BLUE:
        break;
      case YELLOW:
        break;
}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Example

### Use Default Clause as Final `Switch` Clause

Polyspace flags both `switch` statements in this example as noncompliant.

- `switch (i)` contains multiple `case` statements. However, there is no `default` clause to handle cases where `i` is any value other than 0, 1, or 2. Even if `i` can only be 0, 1, or 2, using a `default` clause is defensive programming to catch any circumstance where `i` is unexpectedly a different value.
- `switch (colors)` uses an enumeration for the `switch` variable. However, each value does not have an associated `case` clause. Adding the `case` clause for `case BLUE` makes this `switch` statement compliant.

```
#include <cstdint>

int main(){
    int i;
    //...
    switch (i)        //Noncompliant
    {
    case 0:
        break;
    case 1:
    case 2:
        break;
    }

    enum Colors { RED, BLUE, YELLOW } colors;

    switch (colors) {    //Noncompliant
    case RED:
        break;
    case YELLOW:
        break;
    }

    return 0;
}
```

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-4-7

The condition of a switch statement shall not have bool type

## Description

### Rule Definition

*The condition of a switch statement shall not have bool type.*

### Rationale

Switch statements that have a bool condition might cause confusion or mistakes not caught by the compiler. If statements are better suited to handling bool evaluations. Use If...else statements in place of switch statements that have a bool condition.

### Polyspace Implementation

Polyspace raises this defect whenever a switch-case conditional results in a bool.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Using Bools in Switch Conditions Is Noncompliant

```
#include <cstdint>

int x = 10;

int example(int x)
{
    switch (x > 0) { //Noncompliant
    case true:
        x += 10;
        break;

    case false:
        x -= 10;
        break;

    default:
        x = 0;
        break;
    }

    return x;
}
```

Because the switch statement condition `x > 0` results in a bool, Polyspace marks it as noncompliant.

Use the following if statement in place of the above switch-case:

```
#include <cstdint>

int x = 10;

int example(int x)
{
    if (x > 0) {
        x += 10;
    } else if (x <= 0) {
        x -= 10;
    } else {
        x = 0;
    }

    return x;
}
```

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-4-8

Every switch statement shall have at least one case-clause

## Description

### Rule Definition

*Every switch statement shall have at least one case-clause.*

### Rationale

A switch statement that does not have any case-clauses always results in the default-clause. This behavior is redundant if intentional. If unintentional, a missing case-clause might cause unexpected results. Always provide at least one case-clause to avoid redundancy. If no case-clause is needed, replace or remove the switch statement.

### Polyspace Implementation

Polyspace raises this defect whenever a switch statement contains no case-clause.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Switch Statements Without Case-Clauses Are Noncompliant

```
#include <cstdint>

int x = 10;

int example(int x)
{
    switch (x) { //Noncompliant

    default:
        x = 0;
        break;
    }

    return x;
}
```

Because the switch statement contains only a default clause and no case-clause, Polyspace marks it as noncompliant.

## Check Information

**Group:** Statements
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 6-5-1

A for loop shall contain a single loop-counter which shall not have floating type

## Description

### Rule Definition

*A for loop shall contain a single loop-counter which shall not have floating type.*

### Rationale

A `for` loop without a loop-counter can be replaced with a `while` loop. Use a `while` loop instead in these situations.

Floating-point types are rounded to fit into a finite representation. This causes rounding errors to be introduced with floating-point calculations which can cause unexpected results when performing comparisons.

### Polyspace Implementation

This rule checker flags these situations:

- The `for` loop index has a floating point type.
- More than one loop counter is incremented in the `for` loop increment statement.

  For instance:

  ```
  for(i=0, j=0; i<10 && j < 10;i++, j++) {}
  ```
- A loop counter is not incremented in the `for` loop increment statement.

  For instance:

  ```
  for(i=0; i<10;) {}
  ```

  Even if you increment the loop counter in the loop body, Polyspace still raises a violation. According to the MISRA C++ specifications, a loop counter is one that is initialized in or prior to the loop expression, acts as an operand to a relational operator in the loop expression and *is modified in the loop expression*. If the increment statement in the loop expression is missing, the checker cannot find the loop counter modification and considers as if a loop counter is not present.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Example

### Only Use a Single Loop Counter

Because both loop counters `x` and `i` are incremented, the following example is noncompliant.

```
#include <iostream>

int main()
{
    int x = 0;
    for (int i = 0; i >= x; i = x++)   // Non-compliant
    {
        //...
    }
}
```

### Avoid Using Floating Point Numbers as Loop Counters

In this example, because floating-point calculations can contain rounding errors, the for loop will only execute nine times. This is due to the value of `x` being slightly larger than `1.0f` on the tenth loop.

```
#include <iostream>

int main()
{
    for (float x = 0.1f; x <= 1.0f; x += 0.1f) {   //Noncompliant
        std::cout << x << std::endl;
    }
}
```

## Check Information

**Group:** Statements
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-5-2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=

## Description

### Rule Definition

*If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.*

### Rationale

If the loop counter increments or decrements by more than one per iteration, avoid using == or `!=` in the loop termination condition. It is unclear from visual inspection if such loop termination conditions are ever satisfied.

For instance, if a loop counter `ctr` increments by more than one per iteration, such as `ctr+=2`, and the termination condition is `ctr == someVal`, the counter might skip the value `someVal`, resulting in a nonterminating loop.

Nonterminating loops can result in developer confusion or unexpected values.

### Polyspace Implementation

Polyspace raises this defect whenever all the following are true:

- The loop-counter is not modified by `--` or `++`
- The loop-condition does not contain <=, <, > or >=

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Nonterminating Loops Created by Using Noncompliant Operand Pairings

```
#include <cstdint>

uint32_t row, col;

void example()
{
    for (row = 0; row <= 10; row++) {                 //Compliant
        for (col = 100; col != 10; col -= 4) {        //Noncompliant
            //...
        }
    }
    for (row = 0; row != 10; row++) {            //Compliant
        //...
    }
}
```

Because the second for loop-counter is not modified by `--` or ++, `!=` becomes a noncompliant operand. Use one of the following operands instead: <=, <, > or >=.

### Check Information

**Group:** Statements
**Category:** Required

## Version History

**Introduced in R2013b**

### See Also

`Check MISRA C++:2008 (-misra-cpp)`

#### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 6-5-3

The loop-counter shall not be modified within condition or statement

## Description

### Rule Definition

*The loop-counter shall not be modified within condition or statement.*

### Rationale

The `for` loop has a specific syntax for modifying the loop counter. A code reviewer expects modification using that syntax. Modifying the loop counter elsewhere can make the code harder to review.

### Polyspace Implementation

The checker flags modification of a `for` loop counter in the loop body or the loop condition (the condition that is checked to see if the loop must be terminated).

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 6-5-4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop

## Description

### Rule Definition

*The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.*

### Rationale

If the loop-counter modifier of the loop is not constant, the loop might end unpredictably, causing results contrary to your expectations. Constant increments create predictable loop termination.

### Polyspace Implementation

Polyspace reports this defect whenever the modifier of a loop-counter is not constant. Polyspace also reports this defect when you modify a loop by -=*n* or +=*n* and modify *n* within the loop.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant Nonconstant Loop-Counter Modifiers

```
#include <cstdint>

int i, x = 0;
int y, z = 1;

int ex2()
{
    return z;
}

void example()
{
    for (i = 0; i <= 10; i += x) {          //Noncompliant
        x += 1;
    }

    for (i = 0; i <= 10; i += y) {          //Compliant
        z += i;
    }

    for (i = 0; i >= 10; i -= ex2()) {      //Noncompliant
        z += i;
    }
}
```

Because the first `for` loop uses the loop-counter modifier of `+=x` and modifies the variable `x` within the `for` loop, Polyspace flags the `for` loop as noncompliant.

The second `for` loop shows a compliant case of using a loop-counter modifier of `+=y`, where `y` is not modified within the loop.

The third `for` loop modifies the loop counter by using the return value of a function, `ex2()`. Because the function returns a different value in each iteration, the loop-counter modifier is not constant. Polyspace flags the `for` loop as noncompliant.

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-5-5

A loop-control-variable other than the loop-counter shall not be modified within condition or expression

## Description

### Rule Definition

*A loop-control-variable other than the loop-counter shall not be modified within condition or expression.*

### Rationale

Loops are easier to understand and predict if loop-control-variables, other than loop counters, are not modified within the condition or increment/decrement expression. A volatile typed loop-control-variable is an exception and you can modify it outside of the statement without triggering this violation.

Loop-control-variables are any variables that occur in the loop init-statement, condition, or expression. Loop-control-variables include loop-counters and flags used for early loop termination. A loop-counter is a loop-control-variable that is:

- Initialized prior to the `for` loop or in the init-statement of the `for` loop.
- An operand to a relational operator in the condition of the `for` loop.
- Modified in the expression of the `for` loop.

### Polyspace Implementation

Polyspace raises this defect whenever a loop-counter or flag used for early termination is modified within the condition or expression.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Modifying Loop-Control-Variables Within the Loop Condition

```
#include <cstdint>

bool tf_test, tf_test2 = false;
int32_t x = 0;

bool a_test(bool a)
{
    if (x % 2 == 0)
    {
        a = true;
    }
    else
    {
        a = false;
    }
    return a;
}

void example()
{

    for (x = 0; (x < 10) && (tf_test = a_test(tf_test)); x++)        //noncompliant
    {
        //...
    }

    for (x = 0; (x < 10) && tf_test2; x++)         //compliant
    {
             //...
        tf_test2 = a_test(tf_test2);
    }
}
```

In the first `for` loop, because `tf_test` is a flag used for early loop termination that is modified within the condition, Polyspace flags it as noncompliant.

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-5-6

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool

## Description

### Rule Definition

*A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.*

### Rationale

Loops terminate when the loop-counter value meets a termination condition. You can use additional loop-control-variables if you want to end a loop early.

For instance:

`for(ctr = 0 ; ctr <= 10; ctr++) {…}` terminates when the value of `ctr` is greater than `10`.

`for(ctr = 0 ; ctr <= 10 && level > 0; ctr++) {…}` terminates when the value of `ctr` is greater than `10` or when the value of `level` is greater than `0`.

In the second instance, it is not clear why the condition `level >= 0` terminates the loop early. By using a Boolean variable as a loop-control-variable for early termination, you can use a more descriptive name that reflects the early termination state.

For example:

```
for(ctr = 0 ; ctr <= 10 && fuelTankNotEmpty; ctr++)
{
    /...
    fuelTankNotEmpty = (level >= 0);
}
```

This Boolean variable is called a flag. Boolean flags make loop control logic easier to understand.

### Polyspace Implementation

Polyspace raises this defect whenever a non-Boolean loop-control-variable is modified within the loop statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Modifying Non-Boolean Loop-Control-Variables Within the Loop Statement

```
#include <cstdint>
```

```
int32_t ctr, level = 1;
bool fuelTankNotEmpty = true;

void example()
{
    for(ctr = 0 ; ctr <= 10 && level >= 0; ctr++)        //Noncompliant
    {
        level--;
    }

    for (ctr = 0; ctr <= 10 && fuelTankNotEmpty; ctr++)    //Compliant
    {
        level--;
        fuelTankNotEmpty = (level >= 0);
    }

}
```

In the first `for` loop, because `level` is not a Boolean and is modified within the statement, Polyspace flags it as noncompliant.

The second loop shows how to use a Boolean flag to be compliant with this rule.

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-6-1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement

## Description

### Rule Definition

*Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.*

### Rationale

Using a `goto` statement to jump into nested blocks creates complex control flow, which might cause developer confusion or unexpected results. To avoid unexpected results, place the label the `goto` statement is referring to in the same block or in a block that encloses the `goto` statement.

### Polyspace Implementation

Polyspace raises this defect when the `goto` destination is in a different block than the `goto` statement. This defect is not raised if the `goto` destination is in a block enclosing the `goto` statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using goto Statements to Exit Out of a Block

```
#include <iostream>

int x, y = 0;

void foo1()
{
    int i = 0;
    if (x <= 10) {
        goto err;                    //Noncompliant
    }

    if(x > 10) {
    err:
        std::cout << "Error encountered in loop" << std::endl;
    }

}

void foo2()
{
    for (x = 0; x < 100; ++x) {
        for (y = 0; y < 100; ++y) {
            if (x > y) {
                goto stop;            //Compliant
            }
            //...
        }
    }
stop:
    std::cout << "Error encountered in loop" << std::endl;
}
```

Because the label `err` is located within a separate code block than `goto err`, and that code block does not enclose the code block where `goto err` resides, Polyspace flags the `goto` statement as noncompliant.

The label `stop` is not in the same block as `goto stop`, but is in a block enclosing the `goto stop` statement. This behavior is compliant behavior.

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-6-2

The goto statement shall jump to a label declared later in the same function body

## Description

### Rule Definition

*The goto statement shall jump to a label declared later in the same function body.*

### Rationale

Using a `goto` statement to jump to a label earlier in the same function body creates an iteration. Avoid creating iterations by using `goto` statements. Use iteration statements defined by the core language because they are easier to understand and maintain than goto statements.

### Polyspace Implementation

Polyspace raises this defect when an iteration is formed by using `goto` statements.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Creating Iterations by Using goto Statements

```
#include<iostream>
void foo(int x, int y)
{
test:
    x++;
    if (x <= y) {
        std::cout << x << std::endl;
        goto test;   //Noncompliant
    }
}
```

Because `goto test` sends the code back to the beginning of the function, the goto statement creates a loop. Use looping statements such as `while` or `for` instead.

```
#include<iostream>
void foo(int x, int y)
{
    while (x < y)
    {
        x++;
        std::cout << x << std::endl;
    }
}
```

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 6-6-3

The continue statement shall only be used within a well-formed for loop

## Description

### Rule Definition

*The continue statement shall only be used within a well-formed for loop.*

### Rationale

Using a continue statement can add unnecessary complexity to code, which can cause issues during testing due to the additional logic required.

### Polyspace Implementation

The rule checker reports a violation for the use of continue statements in:

- for loops that are not well-formed, that is, for loops that violate rules 6-5-x
- while loops

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Example

### Only Use continue Statement in Well-Formed for Loops

The for loop in this code is not well-formed as it breaks MISRA C++:2008 Rule 6-5-2. Because the for loop in the example is not well-formed and contains a continue statement, Polyspace raises this violation.

```
#include <cstdint>

uint32_t row, col;

void example()
{
    for (row = 0; row <= 10; row++) {
        for (col = 100; col != 10; col -= 4) {
            //...
            continue;        //Noncompliant
        }
    }
}
```

## Check Information

**Group:** Statements

**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 6-6-4

For any iteration statement there shall be no more than one break or goto statement used for loop termination

## Description

### Rule Definition

*For any iteration statement there shall be no more than one break or goto statement used for loop termination.*

### Rationale

Using multiple `break` or `goto` statements within an iteration statement increases the complexity of the code. For good structured programming, use a single `break` or `goto` statement.

### Polyspace Implementation

Polyspace raises this defect when multiple `break` statements, `goto` statements, or a combination of these statements are present in a single loop.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using Multiple break or goto Statements to Terminate a Loop

```
#include<iostream>
void foo(int x, int y)
{
    for (x = 0; x <= y; x++)
    {
        if (x == y)
        {
            // do this
            break;        //Compliant
        }
        else if (x > y)
        {
            goto error;      //Noncompliant
        }
        else
        {
            //do this
        }
    }

error:
    std::cout << "Error message" << std::endl;
}
```

Because the `if` statement contains a `break` and a `goto` statement, Polyspace flags the statement as noncompliant. In this scenario, Polyspace labels the first `break` or `goto` statement as compliant. Polyspace flags all subsequent `break` or `goto` statements as noncompliant.

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 6-6-5

A function shall have a single point of exit at the end of the function

## Description

### Rule Definition

*A function shall have a single point of exit at the end of the function.*

### Rationale

This rule requires that a `return` statement must occur as the last statement in the function body. Otherwise, the following issues can occur:

- Code following a `return` statement can be unintentionally omitted.
- If a function that modifies some of its arguments has early `return` statements, when reading the code, it is not immediately clear which modifications actually occur.

### Polyspace Implementation

The checker flags these situations:

- A function has more than one `return` statement.
- A non-`void` function has one `return` statement only but the `return` statement is not the last statement in the function.

A `void` function need not have a `return` statement. If a `return` statement exists, it need not be the last statement in the function.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Statements
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 7-1-1

A variable which is not modified shall be const qualified

## Description

### Rule Definition

*A variable which is not modified shall be const qualified.*

### Rationale

Declaring a variable `const` reduces the chances that you modify the variable inadvertently.

### Polyspace Implementation

The checker flags:

- Function parameters or local variables that are not const-qualified but never modified in the function body.
- Pointers that are not const-qualified but point to the same location during its lifetime.

Function parameters of integer, float, enum, and Boolean types are not flagged.

If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. These variables are not flagged.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unmodified Local Variable

```
#include <cstddef>

char getNthChar(const char* str, int N){//Noncompliant
    int index=0;
    while(*(str+index)!='\0'){
        if(index==N)
            return *(str+N);
        ++index;
    }
    return '\0';
}
char getNthChar_const_safe(const char* const str, int N){
    int index=0;

    while(*(str+index)!='\0'){
        if(index==N)
            return *(str+N);
        ++index;
    }
    return '\0';
}
```

In the function `getNthChar()`, the C-string `str` is passed as a `const char*` parameter, meaning that the string `*str` is `const`. Because the pointer `str` does not change value, the pointer itself must be `const` qualified, as shown in the function `getNthChar_const_safe`. Polyspace flags the parameter `str`.

## Check Information

**Group:** Declarations
**Category:** Required

# Version History

**Introduced in R2018a**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 7-1-2

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified

## Description

### Rule Definition

*A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.*

### Polyspace Implementation

The checker flags pointers where the underlying object is not const-qualified but never modified in the function body.

If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. Pointers that point to these variables are not flagged.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Declarations
**Category:** Required

# Version History

**Introduced in R2018a**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 7-2-1

An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.

## Description

### Rule Definition

*An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.*

### Rationale

If your program evaluates an expression with `enum` underlying type and returns a value that is not part of the enumerator list of the enumeration, the behavior is unspecified.

Compliance with this rule validates the assumption made by other rules in this standard that objects of type `enum` contain only values corresponding to the enumerators.

To avoid unspecified behavior when evaluating expressions of type `enum`, use a `switch` statement. For example:

```
#include <cstdio>
enum class E : size_t { Pass, Warning, Error };

E Evaluate(size_t input) noexcept {
    E output = E::Pass;
    switch (input) {
      case 0: {
          output = E::Pass;
          break;
      }
      case 1: {
          output = E::Warning;
          break;
      }
      case 2: {
          output = E::Error;
          break;
      }
      default: {
          output = static_cast<E>(0);
      }
    }
    return output;
}
```

### Polyspace Implementation

Polyspace flags the use of a value that is not part of the enumerators of an enumeration.

Polyspace does not flag the use of unknown values, even if you do not check the range of the value before you use it in the `enum` expression.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Value Does Not Correspond to Enumerators of Enumeration

```
#include <cstdint>

enum class Suit : uint8_t { None = 1, Diamonds, Clubs, Hearts, Spades };

Suit resetRedSuit(uint8_t card) noexcept {
    Suit hand = Suit::None;
    if(card < 6){ //check range of card is inside Suit enum
        hand = static_cast<Suit>(card);
    }

    if (hand == Suit::Diamonds ||
        hand == Suit::Hearts) {
            hand = static_cast<Suit>(0); // Noncompliant
    }
    return hand;
}
```

In this example, Polyspace flags the second `static_cast` in function `resetRedSuit` because the expression uses a value (0) that is not part of the enumerator list of enumerator `Suit`.

## Check Information

**Group:** Declarations
**Category:** Required

# Version History

**Introduced in R2023a**

## See Also

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 7-3-1

The global namespace shall only contain main, namespace declarations and extern "C" declarations

## Description

### Rule Definition

*The global namespace shall only contain main, namespace declarations and extern "C" declarations.*

### Rationale

The rule makes sure that all names found at global scope are part of a namespace. Adhering to this rule avoids name clashes and ensures that developers do not reuse a variable name, resulting in compilation/linking errors, or shadow a variable name, resulting in possibly unexpected issues later.

### Polyspace Implementation

Other than the `main` function, the checker flags all names used at global scope that are not part of a namespace.

The checker does not flag names at global scope if they are declared in `extern "C"` blocks (C code included within C++ code). However, if you use the option `Ignore link errors (-no-extern-c)`, these names are also flagged.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarations
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 7-3-2

The identifier main shall not be used for a function other than the global function main

## Description

### Rule Definition

*The identifier main shall not be used for a function other than the global function main.*

### Rationale

Typically, the `main` function lives in the global namespace and acts as the entry point to a program. The use of `main` in other contexts might defy developer expectations.

### Polyspace Implementation

The rule checker reports a violation if you use the identifier `main` in a namespace other than the global namespace.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Nonstandard Uses of `main`

```
#include <cstdint>

int32_t main() { //Compliant
    //...
    return 0;
}

namespace Backups {
    int32_t main() { //Noncompliant
        //...
        return 0;
    }
}
```

The use of `main` in a namespace other than the global namespace violates the rule.

## Check Information

**Group:** Declarations
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 7-3-3

There shall be no unnamed namespaces in header files

## Description

### Rule Definition

*There shall be no unnamed namespaces in header files.*

### Rationale

According to the C++ standard, names in an unnamed namespace, for instance, `aVar`:

```
namespace {
   int aVar;
}
```

have internal linkage by default. If a header file contains an unnamed namespace, each translation unit with a source file that `#include`-s the header file defines its own instance of objects in the namespace. The multiple definitions are probably not what you intended and can lead to unexpected results, unwanted excess memory usage, or inadvertently violating the one-definition rule.

### Polyspace Implementation

The checker flags unnamed namespaces in header files.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unexpected Results from Unnamed Namespaces in Header Files

Header File: `aHeader.h`

```
namespace { //Noncompliant
   int aVar;
}
```

First source file: `aSource.cpp`

```
#include "aHeader.h"
#include <iostream>

void setVar(int arg) {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = arg;
    std::cout << "Value set at: " << aVar << std::endl;
}
```

Second source file: `anotherSource.cpp`

```
#include "aHeader.h"
#include <iostream>

extern void setVar(int);

void resetVar() {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = 0;
    std::cout << "Value set at: 0" << std::endl;
}

void main() {
    setVar(1);
    resetVar();
}
```

In this example, the noncompliant unnamed namespace leads to two definitions of `aVar` in the translation unit from `aSource.cpp` and the translation unit from `anotherSource.cpp`. The two definitions can lead to possible unexpected output:

```
Current value: 0
Value set at: 1
Current value: 0
Value set at: 0
```

## Check Information

**Group:** Declarations
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 7-3-4

using-directives shall not be used

## Description

### Rule Definition

*using-directives shall not be used.*

### Rationale

`using` directives can inadvertently expose more variables to name lookup than you intend. The wider scope for name lookup increases the likelihood of a variable being unintentionally used. For instance:

```
namespace NS {
    int i;
    int j;
}
using namespace NS;
```

exposes both `NS::i` and `NS::j` to name lookup, but you might have intended to use only one of the variables.

It is preferable to only expose variables that you intend to use or refer to variables by their fully qualified name when you use them. For instance, in the previous example, if you use the variable `NS::i`, expose only this variable to name lookup:

```
using NS::i;
```

or refer to `NS::i` instead of `i` in all instances.

### Polyspace Implementation

The rule checker reports a violation on `using` directives. These directives contain the keyword `using` followed by the keyword `namespace` and the name of a namespace. The `using` directives make all members of a namespace available to the current scope.

The checker does not flag `using` declarations. These declarations also contain the keyword `using` but not the keyword `namespace`. The `using` declarations only expose specific members of a namespace.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Namespace Exposed Through `using` Directive

```
namespace currentDataModel {
    int nodes;
    int count;
}
using namespace currentDataModel; //Noncompliant

namespace lastDataModel {
    int nodes;
    int count;
}
using lastDataModel::count; //Compliant
```

In this example, the `using` directive that exposes all members of the namespace `currentDataModel` violates the rule.

### Check Information

**Group:** Declarations
**Category:** Required

## Version History

**Introduced in R2013b**

### See Also

`Check MISRA C++:2008 (-misra-cpp)`

#### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 7-3-5

Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier

## Description

### Rule Definition

*Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.*

### Rationale

This rule requires that an identifier must not be declared in a namespace *after* an `using` declaration has already introduced an identifier with the same name. The declaration after the `using` statement can lead to developer confusion.

For instance, a `using` declaration such as:

```
using NS::func;
```

introduces the name `func` from the namespace `NS` in the current scope. If there are two overloads of `func` in `NS`, one declared before and another after the `using` statement, only the overload declared before is exposed to name lookup and invoked when `func` is called. However, in a specific call to `func`, a developer might expect the overload declared later to be invoked (perhaps because that overload is a better match based on function arguments).

### Polyspace Implementation

The rule checker reports a violation if an identifier is declared in a namespace after these two events:

1. Another identifier with the same name has been previously declared in the same namespace.
2. A `using` declaration has already exposed that name to name lookup.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Multiple Declarations of Same Name on Either Side of `using` Declaration

```
#include <cstdint>

namespace initialTests {
    void validateInput(uint16_t input);
}

using initialTests::validateInput;

namespace initialTests {
    void validateInput(uint32_t input); //Noncompliant
}

namespace periodicTests {
    void validateResults(uint16_t results);
    void validateResults(uint32_t results); //Compliant
}

using periodicTests::validateResults;

void main() {
    validateInput(0U);
    validateResults(0U);
}
```

In this example:

- The declaration of `validateInput(uint32_t)` in the namespace `initialTests` violates the rule because the name `validateInput` from this namespace has already been exposed with an `using` declaration. This second declaration of `validateInput()` can cause developer confusion. For instance, the call to `validateInput()` in `main` invokes `validateInput(uint16_t)` but a developer might expect `validateInput(uint32_t)` to be invoked because it is a better match for the call.
- The declarations of `validateResults()` in the namespace `periodicTests` do not violate the rule because both declarations appear before the `using` statement. When the function `validateResults()` is called, all overloads of `validateResults()` are exposed to name lookup. A developer can use only the parameters of `validateResults()` in the overload declarations to decide which overload is invoked, thereby reducing chances of confusion.

## Check Information

**Group:** Declarations
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 7-3-6

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files

## Description

### Rule Definition

*using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.*

### Rationale

If `using` directives or declarations are present in a header file, the order in which the header is included might affect the final program results. It is a good practice to ensure that the program behavior does not depend on the order of inclusion of headers.

### Polyspace Implementation

The checker flags `using` directives or declarations in header files.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### `using` Directives in Headers Introduce Dependence on Order of Inclusion

Header File: `headerUnits.h`:

```
void checkValueForOverflows(int32_t);

namespace Units {
    void checkValueForOverflows(int8_t);
}

inline void convertToAcceptedUnit(int8_t originalValue) {
    checkValueForOverflows(originalValue);
}
```

Header File: `headerAcceptedUnits.h`:

```
namespace Units {

}

using namespace Units; //Noncompliant
```

Source file that includes previous two headers:

```
#include <cstdint>
#include "headerUnits.h"
```

```
#include "headerAcceptedUnits.h"

int8_t convert(int8_t meterReading) {
    convertToAcceptedUnit(meterReading);
}
```

In this example, the `using` directive in the header file `headerAcceptedUnits.h` exposes the namespace `Units`. This directive plus the inlined function in the header makes the order of inclusion of `headerUnits.h` and `headerAcceptedUnits.h` important for the final program execution.

- If `headerAcceptedUnits.h` is included before `headerUnits.h`, the function `convertToAcceptedUnit()` invokes the function `checkValueForOverflows(int32_t)`.
- If `headerAcceptedUnits.h` is included after `headerUnits.h`, as shown above, the function `convertToAcceptedUnit()` invokes the function `checkValueForOverflows(int8_t)`.

## Check Information

**Group:** Declarations
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 7-4-2

Assembler instructions shall only be introduced using the asm declaration

## Description

### Rule Definition

*Assembler instructions shall only be introduced using the asm declaration.*

### Rationale

The `asm` declaration is available in all C++ implementations[6] but other ways of introducing assembler instructions might not be available. Using the `asm` declaration makes your code portable across implementations.

### Polyspace Implementation

The rule checker reports uses of methods other than `asm` declarations as rule violations.

For instance, the rule checker allows the `asm` declaration:

```
asm("ADD a0,1")
```

But if the same assembler instructions are introduced through a `#pragma asm`, the checker reports a violation:

```
#pragma asm
   "ADD a0,1"
#pragma endasm
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Introduction of Assembler Instructions Using Nonportable Methods

```
void Delay1 ( void ) {
#pragma asm    //Noncompliant
   "ADD a0,1"
#pragma endasm
     ;
}

void Delay2 ( void ) {
   asm("ADD a0,1"); //Compliant
}
```

The use of `#pragma asm` violates this rule.

[6] How the `asm` declaration is implemented might vary across compilers. See also `AUTOSAR C++14 Rule A7-4-1`.

## Check Information

**Group:** Declarations
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 7-4-3

Assembly language shall be encapsulated and isolated

## Description

### Rule Definition

*Assembly language shall be encapsulated and isolated.*

### Rationale

Assembler instructions can be difficult to port across implementations. Encapsulating and isolating them in a function helps to make your code portable. You can easily track down assembler instructions or provide alternative implementations without changing the rest of the code.

### Polyspace Implementation

The checker flags `asm` statements unless they are encapsulated in a function call.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Assembler Instructions Not Encapsulated in Function Call

```
void DoSomething(void);

void Delay ( void ) {
    asm( "NOP"); //Compliant
    asm( "NOP"); //Compliant
}
void DoSomethingAndDelay (void) {
    DoSomething();
    Delay();// Assembler is encapsulated
    DoSomething();
    asm("NOP"); //Noncompliant
    DoSomething();
}
```

The `asm` statement in the function `DoSomethingAndDelay()` is mixed with regular C code and violates the rule. The `asm` statements in the function `Delay()` are compliant because they form the only instructions in this function. The `Delay()` function thus effectively encapsulates the assembler instructions.

## Check Information

**Group:** Declarations
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 7-5-1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function

## Description

### Rule Definition

*A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.*

### Rationale

Local objects of a function are destroyed at the end of a function call. If you return the address of such an object via a pointer, you might access a destroyed object. For instance:

```
int* foo(int a){
    int b = a^2;
    return &b;
}
```

The function `foo()` returns a pointer to `b`. The variable `b` goes out of scope at the end of the function call and the pointer returned by `foo()` points to a destroyed object. Accessing the pointer leads to undefined behavior.

### Polyspace Implementation

Polyspace raises a violation if a function returns a reference or pointer to a variable that goes out of scope at the end of the function call.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Returning Pointers to Local Variables

```
int gVal;
//...

int* foo(void){
    int val;
    //...
    return &val;//Noncompliant
}
```

In this example, the function `foo()` returns the address of `val`. This local variables goes out of scope at the end of the function call. Accessing the pointer returned by `foo()` results in undefined behavior. Polyspace raises a violation.

## Check Information

**Group:** Declarations
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 7-5-2

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist

## Description

### Rule Definition

*The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.*

### Rationale

If an object continues to point to another object *after* the latter object ceases to exist, dereferencing the first object leads to undefined behavior.

### Polyspace Implementation

The checker flags situations where the address of a local variable is assigned to a pointer defined at global scope.

The checker does not raise violations of this rule if:

- A function returns the address of a local variable. `MISRA C++:2008 Rule 7-5-1` covers this situation.
- The address of a variable defined at block scope is assigned to a pointer that is defined with greater scope, but not global scope.

  For instance:

  ```
   void foobar ( void )
    {
      char * ptr;
      {
        char var;
        ptr = &var;
      }
    }
  ```

  Only if the pointer is defined at global scope is a rule violation raised. For instance, the rule checker flags the assignment here:

  ```
  char * ptr;
  void foobar ( void )
    {
        char var;
        ptr = &var;
    }
  ```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Address of Local Variable Assigned to Global Pointer

```
char * ptr;

void foo (void) {
    char varInFoo;
    ptr = &varInFoo; //Noncompliant
}

void bar (void) {
    char varInBar = *ptr;
}

void main() {
    foo();
    bar();
}
```

The assignment `ptr = &varInFoo` is noncompliant because the global pointer `ptr` might be dereferenced outside the function `foo`, where the variable `varInFoo` is no longer in scope. For instance, in this example, `ptr` is dereferenced in the function `bar`, which is called after `foo` completes execution.

## Check Information

**Group:** Declarations
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 7-5-3

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference

## Description

### Rule Definition

*A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarations
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 7-5-4

Functions should not call themselves, either directly or indirectly

## Description

### Rule Definition

*Functions should not call themselves, either directly or indirectly.*

### Rationale

Variables local to a function are stored in the call stack. If a function calls itself directly or indirectly several times, the available stack space can be exceeded, causing serious failure. Unless the recursion is tightly controlled, it is difficult to determine the maximum stack space required.

### Polyspace Implementation

The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.

You can calculate the total number of recursion cycles using the code complexity metric `Number of Recursions`. Note that unlike the checker, the metric also considers implicit calls, for instance, to compiler-generated constructors during object creation.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Direct and Indirect Recursion

```
void foo1(int);
void foo2(int);

void foo1( int n ) {    // Non-compliant - Indirect recursion foo1->foo2->foo1...
    if(n > 0) {
        foo2(n);
        foo1(n);    // Non-compliant - Direct recursion
        n--;
    }
}

void foo2( int n ) { // Non-compliant - Indirect recursion foo2->foo1->foo2...
    foo1(n);
}
```

In this example, the rule is violated because of:

- Direct recursion `foo1` → `foo1`.

- Indirect recursion `foo1` → `foo2` → `foo1`.
- Indirect recursion `foo2` → `foo1` → `foo2`.

Note that the location of the rule violation is different for direct and indirect recursions:

- When a function calls itself directly, the rule violation is shown on the function call.
- When several functions are involved in an indirect recursion chain, for every function in the chain, a rule violation is shown on the function signature in the function body.

## Check Information

**Group:** Declarations
**Category:** Advisory

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 8-0-1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively

## Description

### Rule Definition

*An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.*

### Rationale

Init-declarator-lists that have multiple declarators might result in ambiguous type assignments and might cause a developer to assign unintended types to certain identifiers. Having a single init-declarator for each declaration clarifies the declaration type and reduces the risk of unwanted type assignments. Avoid multiple declarators in an init-declarator-list or a member-declarator-list.

### Polyspace Implementation

Polyspace flags declarators after the first declarator in an init-declarator-list or a member-declarator-list.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Multiple Declarators Within an Init-Declarator-List

```
#include <cstdint>
#include <string>

class exampleClass{};

void foo() {


    double a, b, c; //Noncompliant

    exampleClass objectOne, objectTwo; //Noncompliant


    int32_t d; int32_t e; //Compliant

    std::string f;  //Compliant


}
```

In this example, Polyspace flags init-declarator-lists that have multiple declarators. For instance:

- `double a, b, c` is noncompliant because the init-declarator-list consists of more than one init-declarator and the `b` and `c` declarators are flagged.
- `exampleClass objectOne, objectTwo` is noncompliant because the init-declarator-list consists of more than one init-declarator and the `objectTwo`declarator is flagged.
- `int32_t d, int32_t e` and `std::string f` are compliant because each init-declarator-list consists of a single init-declarator.

### Check Information

**Group:** Declarators
**Category:** Required

## Version History

**Introduced in R2013b**

### See Also

`Check MISRA C++:2008 (-misra-cpp)`

#### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 8-3-1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments

## Description

### Rule Definition

*Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarators
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 8-4-1

Functions shall not be defined using the ellipsis notation

## Description

### Rule Definition

*Functions shall not be defined using the ellipsis notation.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarators
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 8-4-2

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration

## Description

### Rule Definition

*The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.*

### Polyspace Implementation

The checker detects mismatch in parameter names between:

- A function declaration and the corresponding definition.
- Two declarations of a function, provided they occur in the same file.

  If the declarations occur in different files, the checker does not raise a violation for mismatch in parameter names. Redeclarations in different files are forbidden by `MISRA C++:2008 Rule 3-2-3`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarators
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 8-4-3

All exit paths from a function with non- void return type shall have an explicit return statement with an expression

## Description

### Rule Definition

*All exit paths from a function with non- void return type shall have an explicit return statement with an expression.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarators
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 8-4-4

A function identifier shall either be used to call the function or it shall be preceded by &

## Description

### Rule Definition

*A function identifier shall either be used to call the function or it shall be preceded by &.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarators
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 8-5-1

All variables shall have a defined value before they are used

## Description

### Rule Definition

*All variables shall have a defined value before they are used.*

### Rationale

The C++ standard does not specify what the value of an uninitialized memory is. This value is unpredictable and can be different every time the program runs. Reading and using the value of an uninitialized memory results in unexpected behavior.

### Polyspace Implementation

Polyspace reports a violation of this rule if your code contains these issues:

- `Non-initialized variable`
- `Member not initialized in constructor`
- `Non-initialized pointer`

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Variable Read Before Initialization

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val; //Noncompliant

}
```

### Member Not Initialized in Constructor

In this example, if `flag` is not 0, the member `_c` is not initialized.

The defect appears on the closing brace of the constructor. Following are some tips for navigating in the source code:

- On the **Result Details** pane, see which members are not initialized.
- To navigate to the class definition, right-click a member that is initialized in the constructor. Select **Go To Definition**. In the class definition, you can see all the members, including those members that are not initialized in the constructor.

```
class MyClass {
public:
    explicit MyClass(int);
private:
    int _i;
    char _c;
};

MyClass::MyClass(int flag) {
    if(flag == 0) {
        _i = 0;
        _c = 'a';
    }
    else {
        _i = 1;
    }
}//Noncompliant
```

### Pointer Dereferenced Before Initialization

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j; //Noncompliant

    return pi;
}
```

## Check Information

**Group:** Declarators
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 8-5-2

Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures

## Description

### Rule Definition

*Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.*

### Rationale

The use of nested braces in initializer lists to match the structures of nested objects in arrays, unions, and structs encourages you to consider the order of initialization of complex data types and makes your code more readable. For example, the use of nested braces in the initialization of ex1 makes it easier to see how the nested arrays arr1 and arr2 in struct ex1 are initialized.

```
struct Example
{
    int num;
    int arr1[2];
    int arr2[3];
};

//....
struct Example ex1 {1, {2, 3}, {4, 5, 6}}; //Compliant
```

The rule does not require the use of nested braces if you zero initialize an array, a union, or a struct with nested structures are the top-level, for instance:

```
struct Example ex1 {}; //Compliant
```

### Polyspace Implementation

If you non-zero initialize an array, union, or struct that contains nested structures and you do not use nested braces to reflect the nested structure, Polyspace flags the first element of the first nested structure in the initializer list. For instance, in this code snippet, Polyspace flags the number 2 because it corresponds to the first element of nested structure arr1 inside struct ex1.

```
struct Example
{
    int num;
    int arr1[2];
    int arr2[3];
};

//....
struct Example ex1 {1, 2, 3, 4, 5, 6}; // Non-compliant
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Missing Nested Braces in Initializer of Two-Dimensional Arrays

```
char arr1[2][3] {'a', 'b', 'c', 'd', 'e', 'f'}; //Non-compliant
char arr2[2][3] {{'a', 'b', 'c'}, {'d', 'e', 'f'}}; //Compliant
char arr_top_level[2][3] { }; //Compliant
char arr_sub_level[2][3] { {}, {'d', 'e', 'f'}}; //Non-compliant
```

In this example, two-dimensional array `arr1` is non-compliant because the initializer list does not reflect the nested structure of this array (two arrays of three elements each). The initialization of `arr2` uses nested braces to reflect the nested structure of the array and is compliant. Similarly, the initialization of `arr_top_level` is compliant because it zero initializes the array at the top level. Note that the initialization of `arr_sub_level` is non-compliant because zero-initializes only the first sub-array while explicitly initializing all the elements of the other sub-array.

## Check Information

**Group:** Declarators
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 8-5-3

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized

## Description

### Rule Definition

*In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarators
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 9-3-1

const member functions shall not return non-const pointers or references to class-data

## Description

### Rule Definition

*const member functions shall not return non-const pointers or references to class-data.*

### Rationale

A `const` object cannot be changed post initialization and can only invoke class member functions that are also declared as `const`. These member functions are not expected to change the state of the object.

If a `const` member function returns a non-`const` pointer or reference to a data member of the class, the function can modify the state of an object and violate developer expectations. To avoid this situation, when you define a `const` member function that returns a reference or a pointer to a class data member, specify the return type as `const`.

### Polyspace Implementation

Polyspace flags a violation of this rule only if a `const` member function returns a non-`const` pointer or reference to a non-`static` data member. The rule does not apply to `static` data members.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Specify Return Type of `const` Member Function as `const`

```
class A{

    private:

    int& rInt;
    public:

    int* getR() const{ //Noncompliant
        return &rInt;
    }

    const int* getConstR() const{ //Compliant
        return &rInt;
    }
};
```

In this example:

- The `const` member function `getR()` returns a non-`const` pointer and violates the rule.
- The `const` member function `getConstR()` returns a `const` pointer and is compliant with the rule.

### Check Information

**Group:** Classes
**Category:** Required

## Version History

**Introduced in R2013b**

### See Also

`Check MISRA C++:2008 (-misra-cpp)`

#### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 9-3-2

Member functions shall not return non-const handles to class-data

## Description

### Rule Definition

*Member functions shall not return non-const handles to class-data.*

### Polyspace Implementation

The checker flags a rule violation only if a member function returns a non-`const` pointer or reference to a nonstatic data member. The rule does not apply to static data members.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Classes
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 9-3-3

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const

## Description

### Rule Definition

*If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.*

### Rationale

`const` member functions cannot modify the data members of the class. `static` member function cannot modify the nonstatic data members of the class. If a member function does not need to modify the nonstatic data members of the class, limit their access to data by declaring the member functions as `const` or `static`. Such declaration clearly expresses and enforces the design intent. That is, if you inadvertently attempt to modify a data member through a `const` member function, the compiler catches the error. Without the `const` declaration, this kind of inadvertent error might lead to bugs that are difficult to find or debug.

### Polyspace Implementation

The checker performs these checks in this order:

1. The checker first checks if a class member function accesses a data member of the class. Functions that do not access data members can be declared static.
2. The checker then checks functions that access data members to determine if the function modifies any of the data members. Functions that do not modify data members can be declared const.

A violation on a const member function means that the function does not even access a data member of the class and can be declared static.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Explicitly Restrict Access for Member Functions that Do Not Modify Data Members

```
#include<cstdint>
void Connector(void);
class A
{
public:
    int16_t foo ( ) // Noncompliant
    {
        return m_i;
    }
    int16_t foo2 ( ) // Noncompliant
    {
        Connector();// Might have side-effect
        return m_i;
    }
    int16_t foo3 ( ) // Noncompliant
    {
        return m_s;
    }
    int16_t inc_m ( ) // Compliant
    {
        return ++m_i;
    }
    int16_t& getref()//Noncompliant
    {
        return m_i_ref;
    }
private:
    int16_t m_i;
    static int16_t m_s;
    int16_t& m_i_ref;
};
```

In this example, Polyspace flags the functions `foo`, `foo2`, `foo3`, and `getref` as noncompliant.

- The functions `foo` and `foo3` do not modify any nonstatic data members. Because their data access is not explicitly restricted by declaring them as `const`, Polyspace flags these functions. To fix these defects, declare `foo` and `foo3` as `const`.
- The function `foo2` does not explicitly modify any of the data members. Because it is not declared as `const`, Polyspace flags the function. `foo2` calls the global function `Connector`, which might have side effects. Do not declare `foo2` as a `const` function. In C++11 or later, `const` member functions are expected to be thread-safe, but `foo2` might not be thread-safe because of the side effects of `Connector`. To avoid data races, keep `foo2` as a nonconst function. Justify the defect by using review information or code comments.
- The function `getref` does not modify any data members. Because it is not declared as `const`, Polyspace flags it. Declaring `getref` as `const` resolves this defect, but that is not enough to restrict write access of `getref` because it returns a nonconst reference to `m_i_ref`. To restrict `getref` from modifying `m_i_ref`, the return type of `getref` must also be `const`.

## Check Information

**Group:** Classes
**Category:** Required

# Version History

**Introduced in R2018a**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**
"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 9-5-1

Unions shall not be used

## Description

### Rule Definition

*Unions shall not be used.*

### Rationale

Using unions to store a value might result in misinterpretation of the value and lead to undefined behavior. For instance:

```
union Data{
    int i;
    double d;
};
void bar_int(int);
void bar_double(double);
void foo(void){
    Data var;
    var.d = 3.1416;
    bar_int(var.d);//Undefined Behavior
}
```

In the call to `bar_int`, the `double` data in the union is misinterpreted as an `int`, which is undefined behavior. Compilers might react to this misinterpretation differently depending on their implementation. To avoid undefined behaviors, do not use a `union`.

In some cases, use of unions might be necessary to increase efficiency. In such cases, use unions after documenting the relevant implementation-defined compiler behaviors. In the preceding case, before using a `union`, consult the manual of the compiler that you use and document how the compiler reacts to interpreting a `double` as an `int`.

### Polyspace Implementation

Polyspace flags the declaration of a `union`. You might consider the use of `union` necessary or acceptable in your code. In such cases, justify the violation by annotating the result or by using code comments. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Using union

```
#include <iostream>

union Pi{ //Noncompliant
    int i;
    double d;
};

void foo(void){

    std::cout << std::endl;

    Pi pi;
    pi.d = 3.1416;// pi holds a double
    std::cout << "pi.d: " << pi.d << std::endl;
    std::cout << "pi.i: " << pi.i << std::endl;      // Undefined Behavior

    std::cout << std::endl;

    pi.i = 4;     // pi holds an int
    std::cout << "pi.i: " << pi.i << std::endl;
    std::cout << "pi.d: " << pi.d << std::endl;      // Undefined Behavior

    std::cout << std::endl;

}
```

In this example, the `union Pi` contains a `double` and an `int`. In the code, a `double` is misinterpreted as an `int` and vice versa by using the `union`. These misinterpretations are undefined behaviors and might lead to bugs and implementation dependent code behavior. Polyspace flags the `union` declaration.

## Check Information

**Group:** Classes
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 9-6-2

Bit-fields shall be either bool type or an explicitly unsigned or signed integral type

## Description

### Rule Definition

*Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.*

### Rationale

Using bit-fields require that their underlying bit representations are not implementation-defined. For types other than `bool` and `signed` or `unsigned` integral types, the underlying bit representation is not explicitly known. For instance, the underlying representation of an `int` bit-field can be either signed or unsigned based on implementation. Similarly, ISO/IEC 14882:2003 does not explicitly define the signedness of the underlying bit representation of `wchar_t` types.

Using types other than `bool` and `signed` or `unsigned` integral types as bit fields might result in code that behaves in an implementation-dependent manner and result in bugs that are difficult to diagnose. When using bit fields, use `bool`, `signed` integral types, or `unsigned` integral types.

### Polyspace Implementation

Polyspace reports a violation of this rule if the type of a bit field is:

- An integral type that does not have an explicit sign specification
- A `wchar_t`

Polyspace does not report a violation if the type of a bit field is:

- A `bool`
- An explicitly signed or explicitly unsigned integral type
- An explicitly signed or unsigned `char`

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Specify Signedness of Types When Using Bit Fields

```
#include <cstdint>
struct S
{
    signed int sInt_f : 2; // Compliant
    unsigned int uInt_f : 2; // Compliant
    char Ch_f : 2; // Noncompliant
    signed char sCh_f : 2; // Compliant
    unsigned char uCh_f : 2; // Compliant
    short Sh_f : 2; // Noncompliant
    signed short sSh_f : 2; // Compliant
    unsigned short uSh_f : 2; // Compliant
    int Int_f : 2; // Noncompliant
    bool Bool_f : 2; // Compliant
    wchar_t wch_f : 2; // Noncompliant
    int32_t sInt32_f : 2; // Noncompliant
    int8_t sInt8_f : 2; // Compliant
    long Long_f:2; //Noncompliant
    unsigned long uLong_f:2; //Compliant
};
```

In this example, Polyspace flags the integral type bit fields that are not explicitly signed or unsigned and the `wchar_t` type bit fields.

## Check Information

**Group:** Classes
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 9-6-3

Bit-fields shall not have enum type

## Description

### Rule Definition

*Bit-fields shall not have enum type.*

### Rationale

Using bit fields requires that their underlying bit representations are not implementation-defined. The ISO/IEC 14882:2003 does not explicitly define the signedness of the underlying bit representation of `enum` types. Because the sign of an `enum` type depends on the implementation, the exact number of bits that is required to represent the values in the `enum` is implementation-defined.

To avoid code that behaves differently in different implementations and bugs that are difficult to diagnose, do not use `enum` types as bit fields.

### Polyspace Implementation

Polyspace reports a violation of this rule when you use `enum` types as bit fields.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using `enum` Types in Bit Fields

```
enum Spin {CW, CCW};
void foo(){
    struct DataStruct{
        Spin electron:2; //Noncompliant
    };
}
```

In this example, Polyspace flags the declaration of the object `electron` as a bit field because `electron` is a `enum` type.

## Check Information

**Group:** Classes
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 9-6-4

Named bit-fields with signed integer type shall have a length of more than one bit

## Description

### Rule Definition

*Named bit-fields with signed integer type shall have a length of more than one bit.*

### Rationale

Variables with signed integer bit-field types of length one might have values that do not meet developer expectations. For instance, signed integer types of fixed width such as `std::int16_t` (from `cstdint`) have a two's complement representation. In this representation, a single bit is just the sign bit and the value might be 0 or -1.

### Polyspace Implementation

The checker flags declarations of named variables having signed integer bit field types of length equal to one.

Bit field types of length zero are not flagged.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Compliant and Noncompliant Bit-Field Types

```
#include <cstdint>

typedef struct
{
   std::uint16_t IOFlag :1;    //Compliant - unsigned type
   std::int16_t InterruptFlag :1; //Noncompliant
   std::int16_t Register1Flag :2; //Compliant - Length more than one bit
   std::int16_t : 1; //Compliant - Unnamed
   std::int16_t : 0; //Compliant - Unnamed
   std::uint16_t SetupFlag :1; //Compliant - unsigned type
} InterruptConfigbits_t;
```

In this example, only the second bit-field declaration is noncompliant. A named variable is declared with a signed type of length one bit.

## Check Information

**Group:** Classes
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 10-1-1

Classes should not be derived from virtual bases

## Description

### Rule Definition

*Classes should not be derived from virtual bases.*

### Rationale

The use of virtual bases can lead to many confusing behaviors.

For instance, in an inheritance hierarchy involving a virtual base, the most derived class calls the constructor of the virtual base. Intermediate calls to the virtual base constructor are ignored.

### Polyspace Implementation

Polyspace raises a violation if a class is derived from a `virtual` base class.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use of Virtual Bases

```
class Base {};
class Intermediate: public virtual Base {}; //Noncompliant
class Final: public Intermediate {};
```

In this example, the rule checker raises a violation when the `Intermediate` class is derived from the class `Base` with the `virtual` keyword.

The following behavior can be a potential source of confusion. When you create an object of type `Final`, the constructor of `Final` directly calls the constructor of `Base`. Any call to the `Base` constructor from the `Intermediate` constructor are ignored. You might see unexpected results if you do not take into account this behavior.

## Check Information

**Group:** Derived Classes
**Category:** Advisory

# Version History

**Introduced in R2013b**

## See Also

`MISRA C++:2008 Rule 10-1-2` | `Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 10-1-2

A base class shall only be declared virtual if it is used in a diamond hierarchy

## Description

### Rule Definition

*A base class shall only be declared virtual if it is used in a diamond hierarchy.*

### Rationale

This rule is less restrictive than `MISRA C++:2008 Rule 10-1-1`. Rule 10-1-1 forbids the use of a virtual base anywhere in your code because a virtual base can lead to potentially confusing behavior.

Rule 10-1-2 allows the use of virtual bases in the one situation where they are useful, that is, as a common base class in diamond hierarchies.

For instance, the following diamond hierarchy violates rule 10-1-1 but not rule 10-1-2.

```
class Base {};
class Intermediate1: public virtual Base {};
class Intermediate2: public virtual Base {};
class Final: public Intermediate1, public Intermediate2 {};
```

### Polyspace Implementation

Polyspace flags a class if the class is used as a virtual base in a single and linear hierarchy. In such a hierarchy, `virtual` bases are unnecessary.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using `virtual` Base Classes in Linear Hierarchy

In this example, the classes `B` and `C` in the linear hierarchy `A->B->C` uses virtual base classes. In this hierarchy, using virtual base classes is unnecessary and might cause confusion. Polyspace raises a violation. The class hierarchy `alpha->...->omega` is a diamond hierarchy and in this case, the use of virtual base classes is necessary. Polyspace does not flag this class hierarchy.

```
class A{};
class B: public virtual A{};
class C: public virtual B{}; //Noncompliant

class alpha{};

class beta: public virtual alpha{};
class gamma: public virtual alpha{};
class omega: public beta, public gamma{}; //Compliant
```

## Check Information

**Group:** Derived Classes
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 10-1-3

An accessible base class shall not be both virtual and non-virtual in the same hierarchy

## Description

### Rule Definition

*An accessible base class shall not be both virtual and non-virtual in the same hierarchy.*

### Rationale

The checker flags situations where the same class is inherited as a virtual base class and a non-virtual base class in the same derived class. These situations defeat the purpose of virtual inheritance and causes multiple copies of the base class sub-object in the derived class object.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Base Class Both Virtual and Non-Virtual in Same Hierarchy

```
class Base {};
class Intermediate1: virtual public Base {};
class Intermediate2: virtual public Base {};
class Intermediate3: public Base {};
class Final: public Intermediate1, Intermediate2, Intermediate3 {}; //Noncompliant
```

In this example, the class `Base` is inherited in `Final` both as a virtual and non-virtual base class. The `Final` object contains at least two copies of a `Base` sub-object.

## Check Information

**Group:** Derived Classes
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 10-2-1

All accessible entity names within a multiple inheritance hierarchy should be unique

## Description

### Rule Definition

*All accessible entity names within a multiple inheritance hierarchy should be unique.*

### Rationale

Data members and nonvirtual functions within the same inheritance hierarchy that have the same name might cause developer confusion. The entity the developer intended for use might not be the entity the compiler chooses. Avoid using nonunique names for accessible entities within a multiple inheritance hierarchy.

### Polyspace Implementation

This checker flags entities from separate classes that belong to the same derived class if they have an ambiguous name. The name of an entity is ambiguous if:

- Two variables share the same name, even if they are of different types.
- Two functions share the same name, same parameters, and the same return type.

If the data member accessed in the derived class is ambiguous, Polyspace reports this issue as a compilation issue, not a coding rule violation. The checker does not check for conflicts between entities of different kinds such as member functions against data members.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant Data Members in a Multiple Inheritance Hierarchy

```
#include<iostream>
#include<cstdlib>
#include<cstdint>

using namespace std;

class A
{
public:
    int32_t num;
    int32_t total;
    int32_t sum(int32_t toSum)
    {
        total = toSum + num;
    };
};

class B
{
public:
    int32_t num;
    int32_t total();
    int32_t sum(int32_t toSum)
    {
        num = toSum + num;
    };
};


class C : public A, public B        //Noncompliant
{
public:
    void foo()
    {
        num = total;
        sum(num);
    }
};
```

- Because `class A` and `class B` define their own local variable `int32_t num`, and because `class C` is a multiple inheritance hierarchy containing `class A` and `class B`, Polyspace flags both `int32_t num` variables as noncompliant.
- Because `int32_t sum()` in `class A` and `int32_t sum()` in `class B` share the same name, return type, arguments, and are members of the same multiple inheritance hierarchy, both functions are flagged by Polyspace as noncompliant.
- Because `int32_t total` and `int_32t total()` are different types of class members, Polyspace does not flag them even though they are part of the same multiple inheritance hierarchy.

The ambiguous data members might be reported as compilation issues.

### Nonunique Entity Names When Classes Derive From Templates

Because templates generate code, a class hierarchy deriving from a template might have nonunique names. In this example, the classes `Deriv1` and `Deriv2` derives from two separate instances of the template `myObj`. Then, the class `FinalDerived` derives from both `Deriv1` and `Deriv2`. The template `myObj` generates two version of the function `get()` in the `FinalDerived` class: one from the `myObj<int>` instance and one from the `myObj<short>` instance. The nonunique name might result in unexpected behavior. Polyspace reports a violation

```
template <typename T>
class myObj {
    //...
public:
    void get(void);
};

class Deriv1: public myObj<int> {
public:
    //...
};
```

```
class Deriv2: public myObj<short> {
public:
    //...
};

class FinalDerived: public Deriv1, public Deriv2 { //Noncompliant

public:
    //...
};
```

## Check Information

**Group:** Derived Classes
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 10-3-1

There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy

## Description

### Rule Definition

*There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.*

### Rationale

The checker flags virtual member functions that have multiple definitions on the same path in an inheritance hierarchy. If a function is defined multiple times, it can be ambiguous which implementation is used in a given function call.

### Polyspace Implementation

The checker also raises a violation if a base class member function is redefined in the derived class without the `virtual` keyword.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Virtual Function Redefined in Derived Class

```
class Base {
    public:
      virtual void foo() {
     }
};

class Intermediate1: public virtual Base {
    public:
      virtual void foo() { //Noncompliant
      }
};

class Intermediate2: public virtual Base {
    public:
       void bar() {
          foo();  // Calls Base::foo()
      }
};

class Final: public Intermediate1, public Intermediate2 {
};
```

```
void main() {
    Intermediate2 intermediate2Obj;
    intermediate2Obj.bar(); // Calls Base::foo()
    Final finalObj;
    finalObj.bar(); //Calls Intermediate1::foo()
                    //but you might expect Base::foo()
}
```

In this example, the `virtual` function `foo` is defined in the base class `Base` and also in the derived class `Intermediate1`.

A potential source of confusion can be the following. The class `Final` derives from `Intermediate1` and also derives from `Base` through another path using `Intermediate2`.

- When an `Intermediate2` object calls the function `bar` that calls the function `foo`, the implementation of `foo` in `Base` is called. An `Intermediate2` object does not know of the implementation in `Intermediate1`.
- However, when a `Final` object calls the same function `bar` that calls the function `foo`, the implementation of `foo` in `Intermediate1` is called because of dominance of the more derived class.

You might see unexpected results if you do not take this behavior into account.

To prevent this issue, declare a function as pure virtual in the base class. For instance, you can declare the class `Base` as follows:

```
class Base {
    public:
      virtual void foo()=0;
};

void Base::foo() {
      //You can still define Base::foo()
}
```

## Check Information

**Group:** Derived Classes
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 10-3-2

Each overriding virtual function shall be declared with the virtual keyword

## Description

### Rule Definition

*Each overriding virtual function shall be declared with the virtual keyword.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Derived Classes
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 10-3-3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual

## Description

### Rule Definition

*A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.*

### Rationale

In C++, an abstract class is the base of a polymorphic class hierarchy and the derived classes implement variation of the abstract class. When a virtual function is overridden in a derived class by a pure virtual function, the derived class becomes an abstract class. That a derived class is defined as an abstract class or an implemented function is overridden by a pure virtual function is unexpected behavior, which might confuse a developer.

### Polyspace Implementation

Polyspace flags a pure virtual function if it overrides a function that is not pure virtual.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Do Not Redeclare Functions as Pure Virtual

```
class Conic{
    //...
    public:
    double centerAbscissa;
    double centerOrdinate;
    //..
    virtual double  getArea()=0;
};
class Circle: public Conic{
    //...
    public:
    //...
    double getArea() override{
        //calculate area of circle
    }
};
class Ellipse: public Circle{
    //...
    public:
    //...
    virtual double getArea()=0; //Noncompliant
};
```

In this example, the base class `Conic` is an abstract class because the function `getArea()` is a pure virtual function. The derived class `Circle` implements the function `getArea`. The expectation from such a polymorphic hierarchy is that the virtual function `getArea` calculates the area correctly based on the derived class. When the derived class `Ellipse` redeclares `getArea` as a pure virtual function, the derived class `Ellipse` becomes abstract and the function `Ellipse.getArea()` cannot be invoked. Developers might expect `Ellipse.getArea()` to return the area of the ellipse. Because this redeclaration as a pure virtual function does not meet developer expectation, Polyspace flags the declaration.

## Check Information

**Group:** Derived Classes
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 11-0-1

Member data in non- POD class types shall be private

## Description

### Rule Definition

*Member data in non- POD class types shall be private.*

### Rationale

If classes have data members that are publicly accessible, other classes and functions might interact with the class data members directly. Any change in the class might require updating the clients that use the class. If a class is not a plain-old-data (POD) type, restricting access to its data members enables encapsulation of the class. In such an encapsulated class, the implementation details of the class are opaque to the clients that use it. The class retains control over its implementation and can be maintained independently without impacting the clients that use the class.

### Polyspace Implementation

Polyspace flags nonprivate data members in classes that are not POD types. Polyspace space uses the same definition of POD classes as the standard.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Declare Data Members in Non-POD Classes as `private`

```
class nonPOD{
    nonPOD(){
        //...
    }
    ~nonPOD(){
        //...
    }
    public:
    int getX();
    int setX(int&);
    int getY();
    int setY(int&);
    int getZ();
    int setZ(int&);
    int x; //Noncompliant
    protected:
    int y; //Noncompliant
    private:
    int z;
};
```

In this example, the data members `y` and `z` are not `private`. Polyspace flags them.

## Check Information

**Group:** Member Access Control
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 12-1-1

An object's dynamic type shall not be used from the body of its constructor or destructor

## Description

### Rule Definition

*An object's dynamic type shall not be used from the body of its constructor or destructor.*

### Rationale

The dynamic type of an object is the type of its most derived class. For instance:

```
struct B {
    virtual ~B() {}
};
struct D: B {};
D d;
B* ptr = &d;
```

The dynamic type of the object pointed to by `*ptr` is `D` because that is the most derived class in the polymorphic hierarchy.

When you invoke the dynamic type of a polymorphic object in its constructor or destructor, you might get the type of the constructed or destroyed object instead of the type of the most derived object. This is because when you invoke the dynamic type during construction or destructor, the derived classes might not be constructed yet. Using dynamic types in constructors and destructors might result in unexpected behavior. Calling pure virtual functions from constructors and destructors results in undefined behavior. Avoid using the dynamic type of an object in its constructors or destructors.

### Polyspace Implementation

Polyspace flags these items when they are used in a constructor or a destructor of a polymorphic class:

- The operator `typeid`
- Virtual or pure virtual functions
- The function `dynamic_cast` or implicit C-style casts

Polyspace assumes that a class is polymorphic if it has any virtual member.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using Dynamic Type in Constructors and Destructors

```
#include <cassert>
#include <typeinfo>
```

```
class PS
{
public:
    PS ( )
    {
        typeid ( PS );                // Compliant
    }
};

class PS_1
{
public:
    virtual ~PS_1 ( );
    virtual void bar ( );
    PS_1 ( )
    {
        typeid ( PS_1 );                // Noncompliant
        PS_1::bar ( );                  // Compliant
        bar ( );                      // Noncompliant
        dynamic_cast< PS_1* > ( this ); // Noncompliant
    }
};
```

In this example, class PS has no virtual member. Polyspace does not consider PS a polymorphic class. Because PS is not polymorphic, its dynamic type does not change at run time. Polyspace does not flag using the `typeid` operator in the constructor `PS::PS()`.

`PS_1` is considered polymorphic because it has a `virtual` member function. Because it is polymorphic, its dynamic type changes during run time. Polyspace flags the invocation of its dynamic type in the constructor `PS_1::PS_1()`.

## Check Information

**Group:** Special Member Functions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 12-1-2

All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes

## Description

### Rule Definition

*All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.*

### Rationale

Derived classes that do not explicitly call all base class constructors create ambiguity over which base constructor is used during object construction.

Consider this diamond class hierarchy.

```
            SuperClass
              /    \
       ClassA      ClassB
              \    /
              ClassC
```

When constructing a `ClassC` object, it is unclear whether `ClassA` or `ClassB` is responsible for initializing the `SuperClass`. If the `SuperClass` is constructed with unintended initial values, then the risk of unexpected code behavior increases. Having `ClassC` explicitly specify the constructor used to initialize the `SuperClass` resolves the diamond ambiguity.

To avoid invalid state and unintended initial values, directly call the necessary base class constructors in the derived class constructor initialization list.

### Polyspace Implementation

Polyspace flags the constructor of a derived class if its initialization list:

- Does not explicitly call the constructors of the virtual base classes.
- Does not explicitly call the constructors of the direct nonvirtual base classes.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Class Construction When Using Multiple Inheritance

```
#include <cstdint>

class A {
public:
    A() : a{1} {}
    virtual void abstractA() const = 0;
private:
    int a;
};

class B : public  A {
public:
    B() : b{1} {} //Noncompliant
    void abstractA() const override {}
private:
    int b;
};

class C {
public:
    C() : c{3} {}
private:
    int c;
};

class D : public B, public C {
public:
    D() : B(), C(), e{5} {} //Compliant
private:
    int e;
};

int main() {
    D dName;
    return 0;
}
```

In this example, Polyspace flags the class constructors that do not explicitly initialize the base classes and nonstatic data members. For instance:

- The B class constructor is noncompliant because it is missing an explicit initialization of its base class A. To resolve this issue, call the A class constructor in the B constructor initialization list.
- The D class constructor is compliant because it explicitly initializes both of its direct nonvirtual base classes through initialization list constructor calls.

### Class Construction When Using Multiple and Virtual Inheritance

```
#include <cstdint>

class A {
public:
    A() : a{1} {}
    virtual void abstractA() const = 0;
private:
    int a;
};

class B : public virtual A {
public:
    B() : A(), b{1} {} //Compliant
    void abstractA() const override {}
private:
    int b;
};

class C : public virtual A {
public:
    C() : c{3} {} //Noncompliant
    void abstractA() const override {}
private:
    int c;
};

class D : public B, public C {
public:
    D() : B(), C(), e{5} {} //Noncompliant
private:
    int e;
};

int main() {
    D dName;
    return 0;
}
```

In this example, Polyspace flags the class constructors that do not explicitly initialize the base classes and nonstatic data members. For instance:

- The B class constructor is compliant because it explicitly initializes its direct base class within its initialization list.
- The C class constructor is noncompliant because it does not explicitly call its direct base class A. To resolve this issue, call the A class constructor in the C constructor initialization list.
- The D class constructor is noncompliant because it does not explicitly call its virtual base class A. As a result of multiple and virtual inheritance, the most derived class must initialize the virtual base class. To resolve this issue, call the A class constructor in the D constructor initialization list.

## Check Information

**Group:** Special Member Functions
**Category:** Advisory

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 12-1-3

All constructors that are callable with a single argument of fundamental type shall be declared explicit

## Description

### Rule Definition

*All constructors that are callable with a single argument of fundamental type shall be declared explicit.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Special Member Functions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 12-8-1

A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member

## Description

### Rule Definition

*A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Special Member Functions
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 12-8-2

The copy assignment operator shall be declared protected or private in an abstract class

## Description

### Rule Definition

*The copy assignment operator shall be declared protected or private in an abstract class.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Special Member Functions
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 14-5-1

A non-member generic function shall only be declared in a namespace that is not an associated namespace

## Description

### Rule Definition

*A non-member generic function shall only be declared in a namespace that is not an associated namespace.*

### Rationale

This rule forbids placing generic functions in the same namespace as class (struct) type, enum type, or union type declarations. If the class, enum or union types are used as template parameters, the presence of generic functions in the same namespace can cause unexpected call resolutions. Place generic functions only in namespaces that cannot be associated with a class, enum or union type.

Consider the namespace `NS` that combines a class `B` and a generic form of `operator==`:

```
namespace NS {
   class B {};
   template <typename T> bool operator==(T, std::int32_t);
}
```

If you use class `B` as a template parameter for another generic class, such as this template class `A`:

```
template <typename T> class A {
    public:
      bool operator==(std::int64_t);
}

template class A<NS::B>;
```

the entire namespace `NS` is used for overload resolution when operators of class `A` are called. For instance, if you call `operator==` with an `int32_t` argument, the generic `operator==` in the namespace `NS` with an `int32_t` parameter is used instead of the `operator==` in the original template class `A` with an `int64_t` parameter. You or another developer or code reviewer might expect the operator call to resolve to the `operator==` in the original template class `A`.

### Polyspace Implementation

For each generic function, the rule checker determines if the containing namespace also contains declarations of class types, enum types, or union types. If such a declaration is found, the checker flags a rule violation on the operator itself.

The checker also flags generic functions defined in the global namespace if the global namespace also has class, enum or union declarations.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Generic Operator in Same Namespace as Class Type

```
#include <cstdint>

template <typename T> class Pair {
    std::int32_t item1;
    std::int32_t item2;
    public:
      bool operator==(std::int64_t ItemToCompare);
      bool areItemsEqual(std::int32_t itemValue) {
          return (*this == itemValue);
      }
};

namespace Operations {
    class Data {};
    template <typename T> bool operator==(T, std::int32_t); //Noncompliant
}

namespace Checks {
    bool checkConsistency();
    template <typename T> bool operator==(T, std::int32_t); //Compliant
}

template class Pair<Operations::Data>;
```

In this example, the namespace Operations violates the rule because it contains the class type Data alongside the generic operator==. The namespace Checks does not violate the rule because the only other declaration in the namespace, besides the generic operator==, is a function declaration.

In the method areItemsEqual in template class Pair<Operations::Data>, the == operation invokes the generic operator== method in the Operations namespace. The invocation resolves to this operator== method based on the argument data type (std_int32_t). This method is a better match compared to the operator== method in the original template class Pair.

## Check Information

**Group:** Templates
**Category:** Required

# Version History

**Introduced in R2020b**

## See Also

Check MISRA C++:2008 (-misra-cpp)

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 14-5-2

A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter

## Description

### Rule Definition

*A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Templates
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 14-5-3

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter

## Description

### Rule Definition

*A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.*

### Rationale

When declaring a user-defined assignment operator, the corresponding implicit operator is suppressed. When declaring a template assignment operator that has a generic parameter, this behavior is not preserved. In that case, to suppress the implicit shallow-copying operator, explicitly instantiate a version of the copy assignment operator for the class.

If you do not declare the copy assignment operator for the class, the compiler-generated copy assignment operator might be used instead on implementation. Not declaring a copy assignment operator explicitly might result in an unexpected outcome, such as creating a shallow copy when a deep copy was intended.

### Polyspace Implementation

Polyspace flags this checker if a structure, class, or union contains a template assignment operator that has a generic parameter but no copy assignment operator is present within the structure, class, or union.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant Template Assignment Operator That Has Generic Parameter

```
#include<cstdint>
namespace example
{
  class A  //Noncompliant
  {
    public:

      template <typename T>
      T & operator= ( T const & rhs )
      {
        if ( this != &rhs ) {
          delete i;
          i = new int32_t;
          *i = *rhs.i;
        }
        return *this;
      }
    private:
      int32_t * i;      // Member requires deep copy
  };

  void f ( A const & a1, A & a2 )
  {
    a2 = a1;
  }
};
```

Because no copy assignment operator is declared within the class, Polyspace flags `class A`. The implicitly defined copy assignment operator is not suppressed by the template assignment operator and results in a shallow copy of a1 to a2 when you might want a deep copy.

### Template Assignment Operator That Has a Generic Parameter and Copy Assignment Operator Declared

```
#include<cstdint>
namespace example
{
  class A//Compliant
  {
    public:
      A & operator= (A const & rhs) {};

      template <typename T>
      T & operator= ( T const & rhs )
      {
        if ( this != &rhs ) {
          delete i;
          i = new int32_t;
          *i = *rhs.i;
        }
        return *this;
      }
    private:
      int32_t * i;
  };

  void f ( A const & a1, A & a2 )
  {
    a2 = a1;
  }
};
```

Because this class contains a copy assignment operation declaration, Polyspace does not flag `class A`.

## Check Information

**Group:** Templates
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 14-6-1

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->

## Description

### Rule Definition

*In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->*

### Rationale

When a class template derives from another class template, there might be confusion arising from the use of names that exist in both the base template and the current scope or namespace. When the same name exists in the base class template and a namespace that contains the classes, the scope resolution of these names is dependent on the compiler, which might be contrary to developer's expectation. To avoid confusion, use fully qualified id or `this->` to explicitly disambiguate the intended object when such a name conflict exists.

### Polyspace Implementation

Polyspace flags names for which all of these conditions are true:

- The name exists in the base class.
- The name exists in a namespace that contains the base class.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use Fully Qualified Names in Class Templates That Have Dependent Base Classes

```
typedef signed   int          int32_t;
namespace NS0{
    typedef int32_t TYPE;

    void bar( );
    namespace NS1{
        namespace NS{

            template <typename T>
            class Base;
            template <typename T>
            class Derived : public Base<T>
            {
                void foo ( )
                {
                    TYPE t = 0;                           // Noncompliant
                    bar ( );                                // Noncompliant
                }
                void foo2 ( )
                {
                    NS0::TYPE t1 = 0;                      // Compliant
                    NS0::bar ( );                            // Compliant
                    typename Base<T>::TYPE t2 = 0; // Compliant
                    this->bar ( );                // Compliant
                }
            };
            template <typename T>
            class Base
            {
            public:
                typedef T TYPE;
                void bar ( );
            };
            template class Derived<int32_t>;
        }
    }
}
```

In this example, the names `Type` and `bar` are defined both in the namespace `NS0` and within the class template `Base`. The class template `Derived` derives from `Base`. In `Derived::foo1()`, these names are used without using the fully qualified names or `this->`. It is not clear whether the `TYPE` in `Base::foo1` resolves to `NS0::TYPE` or `Base::TYPE`. You might get different results depending on the implementation of the compiler. Polyspace flags these ambiguous statements.

In `Derived::foo2()`, `TYPE` and `bar` are invoked by using their fully qualified name or `this->`. By using qualified names or `this->`, the ambiguity in scope resolution is bypassed. Polyspace does not flag these uses.

## Check Information

**Group:** Templates
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 14-6-2

The function chosen by overload resolution shall resolve to a function declared previously in the translation unit

## Description

### Rule Definition

*The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.*

### Rationale

In general, you cannot call a function before it is declared, so you expect a function call to resolve to a previously declared function. However, in case of overload resolution of a function call inside a template, this expectation might not be satisfied. The resolution of this overload occurs at the point of template instantiation, not at the point of template definition. So, the call might resolve to a function that is declared *after* the template definition and lead to unexpected results. See examples below.

To satisfy the expectation that a function call *always* resolves to a previously declared function, declare the overloads of a function prior to calling it. Alternatively, use the scope resolution operator `::` or parenthesis to explicitly call a specific previously declared function and bypass the overload resolution mechanism.

### Polyspace Implementation

The checker flags a call to a function or operator in a function template definition if the function or operator is declared *after* the template definition.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Function Call Resolves to Function Declared Later

```
void show (int);

namespace helpers {
    struct params {
        operator int () const;
    };
}

template <typename T> void displayParams(T const & arg) {
    show(arg);    //Non-compliant
    ::show(arg);  //Compliant
    (show)(arg);  //Compliant
}
```

```
namespace helpers {
    void show (params const &);
}

void main() {
    helpers::params aParam;
    displayParams(aParam);
}
```

In this example, the call `show(arg)` in the template `displayParams` resolves to `helpers::show()`, but a developer or code reviewer might not expect this call resolution, since `helpers::show()` is declared later. Polyspace flags this call.

The calls `::show(arg)` and `(show)(arg)` explicitly indicate the previously declared function `show()` declared in the global namespace. Polyspace does not flag these calls.

## Check Information

**Group:** Templates
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 14-7-3

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template

## Description

### Rule Definition

*All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Templates
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 14-8-1

Overloaded function templates shall not be explicitly specialized

## Description

### Rule Definition

*Overloaded function templates shall not be explicitly specialized.*

### Polyspace Implementation

The checker first checks within file scope to find overloads. The checker later looks for call to a specialized template function later. As a result, the checker flags all specializations of overloaded templates even if overloading occurs after the call.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Templates
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 14-8-2

The viable function set for a function call should either contain no function specializations, or only contain function specializations

## Description

### Rule Definition

*The viable function set for a function call should either contain no function specializations, or only contain function specializations.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Templates
**Category:** Advisory

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 15-0-2

An exception object should not have pointer type

## Description

### Rule Definition

*An exception object should not have pointer type.*

### Polyspace Implementation

The checker raises a violation if a `throw` statement throws an exception of pointer type.

The checker does not raise a violation if a NULL pointer is thrown as exception. Throwing a NULL pointer is forbidden by `MISRA C++:2008 Rule 15-1-2`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Exception Handling
**Category:** Advisory

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 15-0-3

Control shall not be transferred into a try or catch block using a goto or a switch statement

## Description

### Rule Definition

*Control shall not be transferred into a try or catch block using a goto or a switch statement.*

### Rationale

Transferring control into a `try` or `catch` block by using a `goto` or a `switch` statement results in ill-formed code that is difficult to understand. The intended behavior of such code is difficult to identify and the code might result in unexpected behavior. Abruptly entering into an exception handling block might cause compilation failure in some compilers while other compilers might not diagnose the issue. To improve code understanding and reduce unexpected behavior, avoid transferring control into a try or a catch block.

### Polyspace Implementation

Polyspace flags the `goto` and `switch` statements that jump into a `try` or a `catch` block.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Jumping into `try` or `catch` Blocks

```
#include<cstdint>
void foo ( int32_t input )
{
    if ( input==1 )
    {
        goto Label_1; // Noncompliant
    }
    if ( input==2 )
    {
        goto Label_2; // Noncompliant
    }
    switch ( input ) //Noncompliant
    {
    case 1:
        try
        {
            Label_1:
        case 2:
            break;
        }
        catch ( ... )
        {
            Label_2:
        case 3:
            break;
        }
        break;
    default:
        {
            //...
            break;
        }
    }
}
```

In this example, `goto` and `switch` statements are used to jump into a `try-catch` block. Jumping into a try-catch block makes the code difficult to understand. Abrupt transfer of control into a `try` block or a `catch` block might result in compilation failure. Polyspace flags the `goto` and `switch` statements.

## Check Information

**Group:** Exception Handling
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 15-1-1

The assignment-expression of a throw statement shall not itself cause an exception to be thrown

## Description

### Rule Definition

*The assignment-expression of a throw statement shall not itself cause an exception to be thrown.*

### Rationale

In C++, you can use a `throw` statement to raise exceptions explicitly. The compiler executes such a `throw` statement in two steps:

- First, it creates the argument for the `throw` statement. The compiler might call a constructor or evaluate an assignment expression to create the argument object.
- Then, it raises the created object as an exception. The compiler tries to match the exception object to a compatible handler.

If an unexpected exception is raised when the compiler is creating the expected exception in a `throw` statement, the unexpected exception is raised instead of the expected one. Consider this code where a `throw` statement raises an explicit exception of class `myException`.

```
class myException{
    myException(){
        msg = new char[10];
        //...
    }
    //...
};

foo(){
    try{
        //..
        throw myException();
    }
    catch(myException& e){
        //...
    }
}
```

During construction of the temporary `myException` object, the `new` operator can raise a `bad_alloc` exception. In such a case, the `throw` statement raises a `bad_alloc` exception instead of `myException`. Because `myException` was the expected exception, the catch block is incompatible with `bad_alloc`. The `bad_alloc` exception becomes an unhandled exception. It might cause the program to abort abnormally without unwinding the stack, leading to resource leak and security vulnerabilities.

Unexpected exceptions arising from the argument of a `throw` statement can cause resource leaks and security vulnerabilities. To prevent such unwanted outcome, avoid using expressions that might raise exceptions as argument in a `throw` statement.

### Polyspace Implementation

Polyspace flags the expressions in `throw` statements that can raise an exception. Expressions that can raise exceptions can include:

- Functions that are specified as `noexcept(false)`
- Functions that contain one or more explicit `throw` statements
- Constructors that perform memory allocation operations
- Expressions that involve dynamic casting

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Expressions That Can Raise Exceptions in `throw` Statements

This example shows how Polyspace flags the expressions in `throw` statements that can raise unexpected exceptions.

```
int f_throw() noexcept(false);

class WithDynamicAlloc {
public:
    WithDynamicAlloc(int n) {
        m_data = new int[n];
    }
    ~WithDynamicAlloc() {
        delete[] m_data;
    }
private:
    int* m_data;
};

class MightThrow {
public:
    MightThrow(bool b) {
        if (b) {
            throw 42;
        }
    }
};

class Base {
    virtual void bar() =0;
};
class Derived: public Base {
    void bar();
};
class UsingDerived {
public:
    UsingDerived(const Base& b) {
        m_d =
        dynamic_cast<const Derived&>(b);
    }
private:
    Derived m_d;
};
class CopyThrows {
public:
    CopyThrows() noexcept(true);
    CopyThrows(const CopyThrows& other) noexcept(false);
};
int foo(){
    try{
        //...
        throw WithDynamicAlloc(10); //Noncompliant
        //...
        throw MightThrow(false);//Noncompliant
        throw MightThrow(true);//Noncompliant
        //...
        Derived d;
        throw  UsingDerived(d);// Noncompliant
        //...
        throw f_throw(); //Noncompliant
        CopyThrows except;
        throw except;//Noncompliant
    }
    catch(WithDynamicAlloc& e){
        //...
    }
    catch(MightThrow& e){
        //...
    }
    catch(UsingDerived& e){
        //...
    }
}
```

- When constructing a `WithDyamicAlloc` object by calling the constructor `WithDynamicAlloc(10)`, exceptions can be raised during dynamic memory allocation. Because the expression `WithDynamicAlloc(10)` can raise an exception, Polyspace flags the `throw` statement `throw WithDynamicAlloc(10);`
- When constructing a `UsingDerived` object by calling the constructor `UsingDervide()`, exceptions can be raised during the dynamic casting operation. Because the expression `UsingDerived(d)` can raise exceptions, Polyspace flags the statement `throw UsingDerived(d)`.
- In the function `MightThrow()`, exceptions can be raised depending on the input to the function. Because Polyspace analyzes functions statically, it assumes that the function `MightThrow()` can raise exceptions. Polyspace flags the statements `throw MightThrow(false)` and `throw MightThrow(true)`.
- In the statement `throw except`, the object `except` is copied by implicitly calling the copy constructor of the class `CopyThrows`. Because the copy constructor is specified as `noexcept(false)`, Polyspace assumes that the copy operation might raise exceptions. Polyspace flags the statement `throw except`
- Because the function `f_throw()` is specified as `noexcept(false)`, Polyspace assumes that it can raise exceptions. Polyspace flags the statement `throw f_throw()`.

## Check Information

**Group:** Exception Handling
**Category:** Required

# Version History

**Introduced in R2020b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 15-1-2

NULL shall not be thrown explicitly

## Description

### Rule Definition

*NULL shall not be thrown explicitly.*

### Rationale

The macro NULL is commonly used to refer to null pointers. Compliers interpret NULL as an integer with value zero, instead of a pointer. When you use NULL explicitly in a throw statement, you might expect the statement to raise a pointer type exception. The throw(NULL) is equivalent to throw(0) and raises an integer exception. This behavior might be contrary to developer expectation and might result in bugs that are difficult to find. Avoid using NULL explicitly in a throw statement.

### Polyspace Implementation

Polyspace flags a throw statement that raises a NULL explicitly. Polyspace does not flag the statement when NULL is raised after casting to a specific type or assigning it to a pointer type.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Do Not Throw NULL Explicitly

```
typedef          char        char_t;
typedef signed   int         int32_t;

#include <cstddef>

void foo()
{
    try {
        char_t * p1 = NULL;
        throw ( NULL );            // Noncompliant
        throw(p1); //Compliant
        throw ( static_cast < const char_t * > ( NULL ) ); // Compliant
    } catch ( int32_t i ) {        // NULL exception handled here
        // /*...*/
    } catch ( const char_t * ) { // Other two exceptions are handled here
        // /*...*/
    }
}
```

In this example, three exceptions are raised directly by using throw statements.

- Polyspace flags the statement throw(NULL) because it explicitly raises NULL as exception. You might expect that this statement raises a pointer type exception that is handled in the second catch block. This statement actually raises an int exception that is handled in the first catch block.
- The other throw statements show the compliant method of using NULL in a throw statement. For instance, the second throw statement raises a char* that is assigned the value NULL. The third throw statement raises a char* by casting NULL to a char*. Because these statements do not raise NULL explicitly, Polyspace does not flag them.

## Check Information

**Group:** Exception Handling
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

Check MISRA C++:2008 (-misra-cpp)

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 15-1-3

An empty throw (throw;) shall only be used in the compound- statement of a catch handler

## Description

### Rule Definition

*An empty throw (throw;) shall only be used in the compound- statement of a catch handler.*

### Rationale

When you use an empty throw statement `(throw;)`, the compiler checks if an exception object is present in the current scope. If the current scope contains an exception object, the compiler raises a temporary object containing the current exception. If the current scope does not contain an exception objects, the compiler invokes `std::terminate()` implicitly. The function `std::terminate()` terminates the program execution in an implementation-defined manner. That is, the exact process of program termination depends on the software and hardware that you are using. For instance, `std:terminate()` might invoke `std::abort()` to abnormally abort the execution without unwinding the stack, leading to resource leak and security vulnerabilities.

The best practice is to use an empty throw statement only in the `catch` block of a `try-catch` construct, which enables you to spread the handling of an exception across multiple catch blocks. Avoid using empty throw statements in scopes that might not contain an exception.

### Polyspace Implementation

Polyspace flags an empty throw statement if it is not within a catch block.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using Empty `throw` Statements Outside `catch` Blocks

```
#include <iostream>
#include <typeinfo>
#include <exception>

void handleException()//function to handle all exception
{
    try {
        throw; // Noncompliant
    }
    catch (std::bad_cast& e) {
        //...Handle bad_cast...
    }
    catch (std::bad_alloc& e) {
        //...Handle bad_alloc...
    }
    catch(...){
        //...Handle other exceptions
    }
}

void f()
{
    try {
        //...something that might throw...
    }
    catch (...) {
        handleException();
    }
}
```

In this example, the function `handleException()` raises the current exception by using an empty throw statement, and then directs it to the appropriate `catch` block. This method of delegating the exception handling works as intended only when the function `handleException()` is called from within a `catch` block. The empty throw statement might cause abrupt termination of the program if the function is called in any other scope that does not contain an exception. Polyspace flags the empty throw statement.

#### Use Empty `throw` Statement to Handle Exceptions in Multiple Blocks

```
#include <iostream>
#include <typeinfo>
#include <exception>
void foo()//function to handle all exception
{
    try {
        //...
    }
    catch (std::bad_cast& e) {
        //...Handle bad_cast...
    }
    catch (std::bad_alloc& e) {
        //...Handle bad_alloc...
    }
    catch(std::exception& e){
        //...Handle std::exceptions
        // if exception cannot be handled
        // throw it again
        throw;//Compliant
    }
}

int main(){
    try{
        foo();
    }
    catch(...){

    }
}
```

This example shows a compliant use of an empty throw statement. The function `foo` contains a `try-catch` construct that handles specific exceptions. If the raised exception cannot be handled, `foo` raises the exception again as an unhandled exception by using an empty throw statement. In `main`, the function `foo` is invoked and any unhandled exception arising from `foo` is handled in a generic `catch(...)` block. By using the empty throw statement, the handling of the exception is spread across the catch blocks of `foo` and `main`. In this case, the empty throw statement is executed only when there is an exception in the same scope because it is within a `catch` block. Polyspace does not flag it.

### Check Information

**Group:** Exception Handling
**Category:** Required

## Version History

**Introduced in R2013b**

### See Also

`Check MISRA C++:2008 (-misra-cpp)`

#### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 15-3-1

Exceptions shall be raised only after start-up and before termination of the program

## Description

### Rule Definition

*Exceptions shall be raised only after start-up and before termination of the program.*

### Rationale

In c++, the process of exception handling runs during execution of `main()`, where exceptions arising in different scopes are handled by exception handlers in the same or adjacent scopes. Before starting the execution of `main()`, the compiler is in startup phase, and after finishing the execution of `main()`, the compiler is in termination phase. During these two phases, the compiler performs a set of predefined operations but does not execute any code.

If an exception is raised during either the startup phase or the termination phase, you cannot write an exception handler that the compiler can execute in those phases. For instance, you might implement `main()` as a `function-try-catch` block to handle exceptions. The `catch` blocks in `main()` can handle only the exceptions raised in `main()`. None of the `catch` blocks can handle exceptions raised during startup or termination phase. When such exceptions are raised, the compiler might abnormally terminate the code execution without unwinding the stack. Consider this code where the construction and destruction of the static object `obj` might cause an exception.

```
class A{
    A(){throw(0);}
    ~A(){throw(0)}
};

static A obj;

main(){
    //...
}
```

The static object `obj` is constructed by calling `A()` before `main()` starts, and it is destroyed by calling `~A()` after `main()` ends. When `A()` or `~A()` raises an exception, no exception handler can be matched with them. Based on the implementation, such an exception can result in program termination without stack unwinding, leading to memory leak and security vulnerabilities.

Avoid operations that might raise an exception in the parts of your code that might be executed before startup or after termination of the program. For instance, avoid operations that might raise exceptions in the constructor and destructor of static or global objects.

### Polyspace Implementation

Polyspace flags global or static variable declaration that uses a callable entity that might raise an exception. For instance:

- Functions: When you call an initializer function or constructor directly to initialize a global or static variable, Polyspace checks whether the function raises an exception and flags the variable declaration if the function might raise an exception. Polyspace deduces whether a function might raise an exception regardless of its exception specification. For instance, If a `noexcept` constructor raises an exception, Polyspace flags it. If the initializer or constructor calls another function, Polyspace assumes the called function might raise an exception only if it is specified as `noexcept(<false>)`. Some standard library functions, such as the constructor of `std::string`, uses pointers to functions to perform memory allocation, which might raise exceptions. Polyspace does not flag the variable declaration when these functions are used.

- External function: When you call external functions to initialize a global or static variable, Polyspace flags the declaration if the external function is specified as `noexcept(<false>)`.
- Virtual function: When you call a virtual function to initialize a global or static variable, Polyspace flags it if the virtual function is specified as `noexcept(<false>)` in any derived class. For instance, if you use a virtual initializer function that is declared as `noexcept(<true>)` in the base class, and `noexcept(<false>)` in a subsequent derived class, Polyspace flags it.
- Pointers to function: When you use a pointer to a function to initialize a global or static variable, Polyspace assumes that pointer to function do not raise exceptions.

Polyspace ignores:

- Exceptions raised in destructors
- Exceptions raised in `atext()` operations

Polyspace also ignores the dynamic context when checking for exceptions. For instance, you might initialize a global or static variable by using function that raises exceptions only in certain dynamic context. Polyspace flags such a declaration even if the exception might never be raised. You can justify such a violation using comments in Polyspace.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Exceptions Before `main()` Starts

This example shows how Polyspace flags construction or initialization of a global or static variable that might raise an exception. Consider this code where static and global objects are initialized by using various callable entities.

```
#include <stdexcept>
#include <string>
class C
{
public:
    C ( ){throw ( 0 );}
    ~C ( ){throw ( 0 );}
};
int LibraryFunc();
int LibraryFunc_noexcept_false() noexcept(false);
int LibraryFunc_noexcept_true() noexcept(true);
int  g() noexcept {
    throw std::runtime_error("dead code");
    return 0;
}
int f() noexcept {
    return g();
}
int init(int a) {
    if (a>10) {
        throw std::runtime_error("invalid case");
    }
    return a;
}
void* alloc(size_t s) noexcept {
    return new int[s];
}
int a = LibraryFunc() +
LibraryFunc_noexcept_true();            // Compliant
int c =
LibraryFunc_noexcept_false() +          // Noncompliant
LibraryFunc_noexcept_true();
static C static_c;                      //Noncompliant
static C static_d;                      //Compliant
C &get_static_c(){
    return static_c;
}
C global_c;                             //Noncompliant
int a3 = f();                           //Compliant
int b3 = g();                           //Noncompliant
int a4 = init(5);                       //Noncompliant
int b5 = init(20);                      //Noncompliant
int* arr = (int*)alloc(5);              //Noncompliant

int main(){
    //...
}
```

- The global pointer arr is initialized by using the function alloc(). Because alloc() uses new to allocate memory, it can raise an exception when initializing arr during the startup of the program. Polyspace flags the declaration of arr and highlights the use of new in the function alloc().
- The integer variable b3 is initialized by calling the function g(), which is specified as noexcept. Polyspace deduces that the correct exception specification of g() is noexcept(false) because it contains a throw() statement. Initializing the global variable b3 by using g() might raise an exception when initializing arr during the startup of the program. Polyspace flags the declaration of b3 and highlights the throw statement in g(). The declaration of a3 by calling f() is not flagged. Because f() is a noexcept function that does not throw, and calls another noexcept function, Polyspace deduces that f() does not raise an exception.
- The global variables a4 and b5 are initialized by calling the function init(). The function init() might raise an exception in certain cases, depending on the context. Because Polyspace deduces the exception specification of a function statically, it assumes that init() might raise an exception regardless of context. Consequently, Polyspace flags the declarations of both a4 and b5, even though init() raises an exception only when initializing b5.
- The global variable global_int is initialized by calling two external functions. The external function LibraryFunc_noexcept_false() is specified as noexcept(false) and Polyspace assumes that this external function might raise an exception. Polyspace flags the declaration of global_int. Polyspace does not flag the declaration of a because it is initialized by calling external functions that are not specified as noexcept(false).

- The static variable `static_c` and the nonstatic global variable `global_c`is declared and initialized by using the constructor of the class `C`, which might raise an exception. Polyspace flags the declarations of these variables and highlights the `throw()` statement in the constructor of class `C`. Polyspace does not flag the declaration of the unused static variable `static_d`, even though its constructor might raise an exception. Because it is unused, `static_d` is not initialized and its constructor is not called. Its declaration does not raise any exception.

## Check Information

**Group:** Templates
**Category:** Required

# Version History

**Introduced in R2020b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 15-3-2

There should be at least one exception handler to catch all otherwise unhandled exceptions

## Description

### Rule Definition

*There should be at least one exception handler to catch all otherwise unhandled exceptions.*

### Polyspace Implementation

The checker reports a violation if any of these conditions are true:

- There is no `try`/`catch` in the `main` function.
- The `catch` block does not include a catch-all (`catch(...)`) handler block.

The rule is not checked if a `main` function does not exist.

The checker does not determine if an exception of an unhandled type eventually propagates to `main`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Exception Handling
**Category:** Advisory

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 15-3-3

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases

## Description

### Rule Definition

*Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.*

### Rationale

The handler `catch` blocks of a function `try` block handle exception that are raised from the body of the function and the initializer list. When used in class constructors and destructors, these `catch` blocks might handle exceptions that arise during the creation or destruction of the class nonstatic members. That is, the `catch` blocks might be executed before or after the lifetime of the nonstatic members of a class. If the nonstatic members of a class are accessed in such `catch` blocks, the compiler might attempt to access objects that are not created yet or already deleted, which is undefined behavior. For instance:

```
class C{

    private:
    int* inptr_x;
    public:
    C() try: inptr_x(new int){}
    catch(...){
        intptr_x = nullptr;
        //...
    }
};
```

Here, the constructor of `C` is implemented by using a function `try` block to handle any exception arising from the memory allocation operation in the initializer list. In the `catch` block of this function-`try` block, the class member `C.intptr_x` is accessed. The catch block executes when the memory allocation for `intptr_x` failed. That is, the catch block attempts to access the member before its lifetime, which is undefined behavior.

To avoid undefined behavior, avoid using the nonstatic data members or base classes of an object in the catch block of the function-try-block implementation of its constructors and destructor.

### Polyspace Implementation

If a statement in the catch block of a constructor or destructor function-`try` block accesses any of these, Polyspace flags the statement:

- The nonstatic members of the object
- The base classes of the object
- The nonstatic members of the base classes

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Accessing Nonstatic Members of Classes in `function-trycatch` Blocks

```
#include<cstdint>
class B
{
public:
    B ( ) try: x(0){/*...*/}
    catch ( ... )
    {
        if ( 0 == x ){/*...*/}  //Noncompliant
        //...
    }
    ~B ( ) try{/*...*/}
    catch ( ... )
    {
        if ( 0 == x ){/*...*/} //Noncompliant
        //...
        else if (sb == 1){/*...*/} //Compliant
        //....
    }
public:
    static int32_t sb;
protected:
    int32_t x;
};

class D : public B
{
public:
    D ( ) try: B(),y{0}{/*...*/}
    catch ( ... )
    {
        if ( 0 == x ){/*...*/}  //Noncompliant
        //...
        else if (y == 1){/*...*/}   //Noncompliant
        //...
    }
    ~D ( )try {/*...*/}
    catch ( ... )
    {
        if ( 0 == x ) {/*...*/} //Noncompliant
        //...
    }
protected:
    int32_t y;
};
```

In this example, the constructors and destructors of `B` and `D` are implemented by using function-try blocks. The `catch` blocks of these function-`try` blocks access the nonstatic members of the class and its base class. Polyspace flags accessing these nonstatic members in the `catch` blocks. Because the lifetime of `static` members is greater than the lifetime of the object itself, Polyspace does not flag accessing `static` objects in these `catch` blocks.

## Check Information

**Group:** Exception Handling
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 15-3-4

Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point

## Description

### Rule Definition

*Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.*

### Rationale

In C++, when an operation raises an exception, the compiler tries to match the exception with a compatible exception handler in the current and adjacent scopes. If no compatible exception handler for a raised exception exists, the compiler invokes the function `std::terminate()` implicitly. The function `std::terminate()` terminates the program execution in an implementation-defined manner. That is, the exact process of program termination depends on the particular set of software and hardware that you use. For instance, `std::terminate()` might invoke `std::abort()` to abnormally abort the execution without unwinding the stack. If the stack is not unwound before program termination, then the destructors of the variables in the stack are not invoked, leading to resource leak and security vulnerabilities.

Consider this code where multiple exceptions are raised in the try block of code.

```
class General{/*...  */};
class Specific : public General{/*...*/};
class Different{}
void foo() noexcept
{
    try{
        //...
        throw(General e);
        //..
        throw( Specific e);
        // ...
        throw(Different e);
    }
    catch (General& b){

    }
}
```

The catch block of code accepts references to the base class `General`. This catch block is compatible with exceptions of the base class `General` and the derived class `Specific`. The exception of class `Different` does not have a compatible handler. This unhandled exception violates this rule and might result in resource leaks and security vulnerabilities.

Because unhandled exceptions can lead to resource leak and security vulnerabilities, match the explicitly raised exceptions in your code with a compatible handler.

### Polyspace Implementation

- Polyspace flags a `throw` statement in a function if a compatible catch statement is absent in the call path of the function. If the function is not specified as `noexcept`, Polyspace ignores it if its call path lacks an entry point like `main()`.
- Polyspace flags a `throw` statement that uses a `catch(â€¦)` statement to handle the raised exceptions.
- Polyspace does not flag rethrow statements, that is, `throw` statements within catch blocks.
- You might have compatible catch blocks for the `throw` statements in your function in a nested try-catch block Polyspace ignores nested try-catch blocks. Justify `throw` statements that have compatible catch blocks in a nested structure by using comments. Alternatively, use a single level of try-catch in your functions.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Match `throw` Statements with Compatible Catch Blocks

This example shows how Polyspace flags operations that raise exceptions without any compatible handler. Consider this code.

```
#include <stdexcept>

class MyException : public std::runtime_error {
public:
    MyException() : std::runtime_error("MyException") {}
};

void ThrowingFunc() {
    throw MyException(); //Noncompliant
}

void CompliantCaller() {
    try {
        ThrowingFunc();
    } catch (std::exception& e) {
        /* ... */
    }
}

void NoncompliantCaller() {
    ThrowingFunc();
}

int main(void) {
    CompliantCaller();
    NoncompliantCaller();
}

void GenericHandler() {
    try {
        throw MyException(); //Noncompliant
    } catch (...) {
        /* ... */
    }
}

void TrueNoexcept() noexcept {
    try {
        throw MyException();//Compliant
    } catch (std::exception& e) {
        /* ... */
    }
}

void NotNoexcept() noexcept {
    try {
        throw MyException(); //Noncompliant
    } catch (std::logic_error& e) {
        /* ... */
    }
}
```

- The function `ThrowingFunc()` raises an exception. This function has multiple call paths:

  - `main()->CompliantCaller()->ThrowingFunc()`: In this call path, the function `CompliantCaller()` has a catch block that is compatible with the exception raised by `ThrowingFunc()`. This call path is compliant with the rule.
  - `main()->NoncompliantCaller()->ThrowingFunc()`: In this call path, there are no compatible handlers for the exception raised by `ThrowingFunc()`. Polyspace flags the `throw` statement in `ThrowingFunc()` and highlights the call path in the code.

  The function `main()` is the entry point for both of these call paths. If `main()` is commented out, Polyspace ignores both of these call paths. If you want to analyze a call path that lacks an entry point, specify the top most calling function as `noexcept`.
- The function `GenericHandler()` raises an exception by using a `throw` statement and handles the raised exception by using a generic catch-all block. Because Polyspace considers such catch-all handler to be incompatible with exceptions that are raised by explicit `throw` statements, Polyspace flags the `throw` statement in `GenericHandler()`.
- The `noexcept` function `TrueNoexcept()` contains an explicit `throw`statement and a catch block of compatible type. Because this `throw` statement is matched with a compatible `catch` block, it is compliant with the rule.
- The `noexcept` function `NotNoexcept()` contains an explicit `throw` statement, but the catch block is not compatible with the raised exception. Because this `throw` statement is not matched with a compatible catch block, Polyspace flags the `throw` statement in `NotNoexcept()`.

## Check Information

**Group:** Exception Handling
**Category:** Required

# Version History

**Introduced in R2020b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 15-3-5

A class type exception shall always be caught by reference

## Description

### Rule Definition

*A class type exception shall always be caught by reference.*

### Rationale

If a class type exception is caught by value, the exception object might be sliced. For instance:

```
class baseException();
class derivedException : public baseException {};

void foo() {
    try {
        //...
        throw derivedException();
    }
    catch (baseException e) { //slices the thrown exception
        //...
    }
}
```

When the catch block in `foo()` catches the `derivedException` object, you might expect the object to remain a `derivedException` object. Because the object is caught by value, it is sliced to a `baseException` object. Unintended object slicing risks unexpected code behavior at runtime. To avoid object slicing, catch class type exceptions by reference or `const` reference.

### Polyspace Implementation

Polyspace flags catch statements where class type exceptions are caught by value.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Catch Exceptions by Reference

```
#include <exception>
#include <iostream>

class baseException : public std::exception {
public:
    baseException() : exception() {}
    const char* what() const noexcept(true) override {
        return "Base Exception Object";
    }
};

class derivedException : public baseException {
public:
    derivedException() : baseException() {}
    const char* what() const noexcept(true) override {
        return "Derived Exception Object";
    }
};

class exampleException{};

void foo() {
    try {
        throw derivedException();
    }
    catch (baseException e) { //Noncompliant
        std::cout << e.what();
    }
    catch (derivedException e) { //Noncompliant
        std::cout << e.what();
    }
    catch (exampleException e) { //Noncompliant
    }
    catch (baseException &e) { //Compliant
        std::cout << e.what();
    }
    catch (const baseException &e) { //Compliant
        std::cout << e.what();
    }
    catch (derivedException &e) { //Compliant
        std::cout << e.what();
    }
    catch (const derivedException &e) { //Compliant
        std::cout << e.what();
    }
}
```

In this example, Polyspace flags the `catch` blocks that catches exception objects by value. For instance:

- Catch blocks for exceptions of type `baseException`, `derivedException`, and `exampleException` are noncompliant because the thrown class type exception is caught by value. These blocks might slice the exception objects.
- Catch blocks for exceptions of type `baseException &`, `const baseException &`, `derivedException &`, and `const derivedException &` are compliant because the class type exception is caught by reference or `const` reference.

## Check Information

**Group:** Exception Handling
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 15-3-6

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class

## Description

### Rule Definition

*Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.*

### Rationale

In a `try-catch` or `function-try` block, exception objects of a derived class match to handler `catch` blocks that accept the base class. If you place handlers of the base exception class before handlers of the derived exception class, the base class handler handles both base and derived class exceptions. The derived class handler becomes unreachable code, which is unexpected behavior. When using a class hierarchy to raise exceptions, make sure that the handler of a derived class precedes the handler of a base class.

### Polyspace Implementation

Polyspace flags a handler block if it follows a handler of a base class.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Order Handler Blocks from most Derived to Base Class

```
#include<exception>
// classes used for exception handling
class MathError { };
class NotANumber: public MathError { };
class DivideByZero: public NotANumber{};

void bar(void){
    try
    {
        // ...
    }
    catch ( MathError &e )
    {
        // ...
    }
    catch ( NotANumber &nan ) // Noncompliant
    {
        // Unreachable Code
    }
    catch (DivideByZero &dbz)//Noncompliant
    {
        //Unreachable Code
    }
}
```

In this example, three classes in a hierarchy might arise in the `try` block. The handler `catch` blocks handle the exceptions.

- The block `catch ( NotANumber &nan )` follows the handler of its base class `catch ( MathError &e )`. Because the exception of class `NotANumber` also matches to the handler `catch ( MathError &e )`, the handler block `catch ( NotANumber &nan )` becomes unreachable code. The order of this block is noncompliant with this rule. Polyspace flags the handler block.
- The block `catch ( DivideByZero &dbz )` becomes unreachable code because exceptions of the class `DivideByZero` match to the preceding handlers of its base classes. Polyspace flags the handler block `catch ( DivideByZero &dbz )`.

## Check Information

**Group:** Exception Handling
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 15-3-7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last

## Description

### Rule Definition

*Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.*

### Rationale

In a try-catch statement or function-try block, the compiler matches the raised exception with a `catch()` handler. The `catch(â€¦)` handler matches any exception. Handlers after the catch-all handler within the same try-catch statement or function try-block are ignored by the compiler during the exception handling process and are unreachable code.

Having a handler after the catch-all handler might result in developer confusion as to why certain intended handlers are not being executed. Likewise, the catch-all handler might not handle the exception in the way the developer intends, resulting in confusion.

### Polyspace Implementation

Polyspace raises this defect whenever a handler appears after the catch-all handler within the try-catch statement or function try-block.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Handlers After the Catch-All Handler Are Noncompliant

```
#include <iostream>
#include <exception>

using namespace std;

int main()
{

    try
    {
        //some code
    } catch(exception& e1) {     //Compliant

        //...

    } catch(...) {               //Compliant
```

```
        //...

    } catch(exception& e2) {     //Noncompliant

        //...

    }
    }

    return 0;
}
```

Because the `catch (exception& e2)` handler comes after the `catch(â€¦)` handler, Polyspace flags the handler before the catch-all handler as noncompliant. This issue might result in a compilation error.

### Check Information

**Group:** Exception Handling
**Category:** Required

## Version History

**Introduced in R2013b**

### See Also

`Check MISRA C++:2008 (-misra-cpp)`

#### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 15-4-1

If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis . See “Checkers Deactivated in Polyspace as You Code Analysis”.*

### Rule Definition

*If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.*

### Rationale

Declaring a function with different exception specification in different source files leads to undefined behavior.

### Polyspace Implementation

Polyspace reports a violation is there is a mismatch in the exception specification of a function in different source files.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Mismatched Exception Specification

In this example, the function `foo()` is specified to raise `std::bad_alloc` exceptions in `src1.cpp`. In another file `src2.cpp`, `foo()` is specified to raise `std::exception`, perhaps inadvertently. The mismatch leads to undefined behavior in the function `bar()`. Polyspace reports a violation of this rule.

```
//src1.cpp
#include<new>
void foo() throw(std::bad_alloc){
    try{
    int *p1 = new int[50];
    }catch(...){
        //....
        std::bad_alloc e;
        throw e;
    }
}
```

```
//src2.cpp
#include<new>
#include<stdexcept>
extern bool condition();
extern void foo() throw(std::exception);//Noncomplian
void bar() {
    foo();

}
```

## Check Information

**Group:** Exception Handling
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 15-5-1

A class destructor shall not exit with an exception

## Description

### Rule Definition

*A class destructor shall not exit with an exception.*

### Polyspace Implementation

The checker flags exceptions thrown in the body of the destructor. If the destructor calls another function, the checker does not detect if that function throws an exception.

The checker does not detect these situations:

- A `catch` statement does not catch exceptions of all types that are thrown.

  The checker considers the presence of a `catch` statement corresponding to a `try` block as indication that an exception is caught.
- `throw` statements inside `catch` blocks

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Exception Handling
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 15-5-2

Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s)

## Description

### Rule Definition

*Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).*

### Polyspace Implementation

The checker flags situations where the data type of the exception thrown does not match the exception type listed in the function specification.

For instance:

```
void goo ( ) throw ( Exception )
  {
    throw 21; // Non-compliant - int is not listed
  }
```

The checker limits detection to `throw` statements that are in the body of the function. If the function calls another function, the checker does not detect if the called function throws an exception.

The checker does not detect `throw` statements inside `catch` blocks.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Exception Handling
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 15-5-3

The terminate() function shall not be called implicitly

## Description

### Rule Definition

*The terminate() function shall not be called implicitly.*

### Polyspace Implementation

The checker flags situations that might result in calling the function `std::terminate()` implicitly. These situations might include:

- An exception escapes uncaught. This also violates `MISRA C++:2008 Rule 15-3-2`. For instance:
  - Before an exception is caught, it escapes through another function that throws an uncaught exception. For instance, a catch statement or exception handler invokes a copy constructor that throws an uncaught exception.
  - An empty `throw` expression raises an uncaught exception again.
- A class destructor raises an exception. Exceptions in destructors also violates `MISRA C++:2008 Rule 15-5-1`.
- A termination handler that is passed to `std::atexit` raises an unhandled exception.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Unhandled Exceptions

```
#include <stdexcept>
#include <new>
class obj
{
public:
    obj() noexcept(false){}
    obj(const obj& a){
        //...
        throw -1;
    }
    ~obj()
    {
        try{
            // ...
            throw std::runtime_error("Error2"); // Noncompliant
        }catch(std::bad_alloc& e){
        }
    }
};
obj globalObject;
void atexit_handler(){//Noncompliant
    throw std::runtime_error("Error in atexit function");
}
void main(){//Noncompliant
    try{
        //...
        obj localObject = globalObject;
        std::atexit(atexit_handler);
    }catch(std::exception& e){

    }
}
```

In this example, Polyspace flags unhandled exceptions because they result in implicit calls to `std::terminate()`.

- The destructor `~obj()` does not catch the exception raised by the `throw` statement. The unhandled exception in the destructor results in abrupt termination of the program through an implicit call to `std::terminate`. Polyspace flags the `throw` statement in the destructor of `obj`.
- The `main()` function does not handle all exceptions raised in the code. Because an unhandled exception might result in an implicit call to `std::terminate()`, Polyspace flags the `main()` function.
- The termination handler `atexit_handler` raises an uncaught exception. The function `atexit_handler` executes after the main finishes execution. Unhandled exceptions in this function cannot be handled elsewhere, leading to an implicit call to `std::terminate()`. Polyspace flags the function.

## Check Information

**Group:** Exception Handling
**Category:** Required

# Version History

**Introduced in R2018a**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 16-0-1

#include directives in a file shall only be preceded by other preprocessor directives or comments

## Description

### Rule Definition

*#include directives in a file shall only be preceded by other preprocessor directives or comments.*

### Rationale

Grouping all #include preprocessor directives at the beginning of the source file makes the code more readable. #include directives might include header files where macros are defined. If you use such a macro before including its definition, you might encounter unexpected code behavior.

### Polyspace Implementation

Polyspace raises this defect when an `#include` directive comes after any code that is not a comment or preprocessor directive. Polyspace ignores code that is hidden by using conditional compilation directives such as `#if` or `#ifdef`. Polyspace does not report a violation of this rule when an `#include` directive is located within an `extern "C"` block.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### #include Directive Preceded by Noncompliant Code

```
//this comment is compliant     //Compliant
/*
    This comment is compliant
*/

#ifndef TESTING_H               //Compliant
#define TESTING_H               //Compliant

#include <iostream>             //Compliant
using namespace std;            //Compliant
#include <exception>            //Noncompliant

#endif
```

Because an include directive follows a code statement that is neither a preprocessor directive nor a comment, `Polyspace` flags the include directive.

## Check Information

**Group:** Preprocessing Directives

**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 16-0-2

Macros shall only be #define 'd or #undef 'd in the global namespace

## Description

### Rule Definition

*Macros shall only be #define 'd or #undef 'd in the global namespace.*

### Rationale

If you define or undefine macros in a local namespace, you might expect the macro to be valid only in the local namespace. But macros do not follow the scoping mechanism. Instead, the compiler replaces all occurrences of a macro by its defined value beginning at the `#define` statement until the end of file or until the macro is redefined. This behavior of macros might be contrary to developer expectation and might cause logic errors that result in bugs.

### Polyspace Implementation

Polyspace flags a `#define` or `#undef` statement that is placed within a block instead of in the global namespace.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Macros in Local Namespaces

```
#include<cstdlib>
#define HCUT 1
namespace unnormalized{
    #define HCUT 6582 //Noncompliant
    void foo(){
        //...
    }
};
void bar(){
    int intEnergy = HCUT*10;
    //HCUT is 6582, you might expect HCUT=1;
}

namespace uniteV{
    const double hcut = 6582; //eV
    void foo(){

    }
};
```

In this example, different values of `HCUT` are defined, perhaps to accommodate code written by using different systems of unit. You might expect the definition of `HCUT` in the namespace `unnormalized` to remain limited to the namespace. But the value of HCUT remains 6582 until the end of file. For instance, in the function bar, you might expect that HCUT is one, but the value of HCUT remains 6582, which might cause logic error, unexpected results, and bugs. Polyspace flags the #define statement within the local namespace.

To implement constants that might have different values in different scopes, use const variables, as shown in the namespace uniteV. Avoid using macros to represent constants that might require different values in different scopes.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

Check MISRA C++:2008 (-misra-cpp)

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 16-0-3

#undef shall not be used

## Description

### Rule Definition

*#undef shall not be used.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 16-0-4

Function-like macros shall not be defined

## Description

### Rule Definition

*Function-like macros shall not be defined.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 16-0-5

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives

## Description

### Rule Definition

*Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.*

### Rationale

When a compiler encounters function-like macros, it replaces the argument of the macro into the replacement code. If the argument contains a token that looks like preprocessing directives, the replacement process during macro expansion is undefined. Depending on the environment, such a function-like macro might behave in unexpected ways, leading to errors and bugs.

### Polyspace Implementation

Polyspace flags calls to function-like macros if their argument starts with the character #.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

### Avoid Arguments That Start with # in Function-Like Macros

```
#include<cstdlib>
#include<iostream>
#define PRINT(ARG) std::cout<<#ARG
//....
#define Error1
//...


void foo(void){
    PRINT(
    #ifdef Error1  //Noncompliant
    "Error 1"
    #else
    "Error 2"
    #endif  //Noncompliant
    );

}
```

In this example, the function-like macro `PRINT` is invoked with an argument that chooses between two strings by using an `#ifdef` block. Depending on the environment, the output of this code might be `#ifdef Error1 //Noncompliant "Error 1" #else "Error 2" #endif // Noncompliant` or `Error 1`. Polyspace flags the arguments that start with the character #.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 16-0-6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##

## Description

### Rule Definition

*In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.*

### Rationale

When you invoke function-like macros, the compiler expands the macro by replacing its parameters with the tokens. Then the compiler substitutes the expanded macro into the code. This expansion and substitution process does not take precedence of operation into account. The function-like macros might produce unexpected results if their parameters are not enclosed in parenthesis. For instance, consider this function-like macro:

```
#define dustance_from_ten(x) x>10? x-10:10-x
```

The macro is intended to measure the distance of a number from ten. When you invoke the macro with the argument (a-b), the macro expands to:

```
a-b>10: a-b-10:10-a-b
```

The expression `10-a-b` is equivalent to `10-(a+b)` instead of the intended distance `10-(a-b)`. This unexpected behavior might result in errors and bugs. To avoid such unexpected behaviors, enclose parameters of a function-like macro in parentheses.

The exception to this rule is when a parameter is used as an operand of # or ##.

### Polyspace Implementation

Polyspace flags function-like macro definitions if the parameters are not enclosed in parenthesis. Polyspace does not flag unparenthesized parameters if they are preceded by the operators `.`, `->`, or the characters #, ##.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Enclose Parameters of Function-Like Macros in Parentheses

```
#include<iostream>
#include<cmath>
#define abs(x) (x>0) ? x:-x //Noncompliant

double foo(double num1, double num2){
    return log(abs(num1-num2));
}

int main(){
    std::cout<<foo(10,10.5);
}
```

In this example, when you invoke `foo(10,10.5)`, you might expect the output to be `log(0.5)` or `-0.69`. Because the parameters of `abs` are not enclosed in parentheses, the output becomes `log(-20.5)` or `NaN`, which is unexpected and might lead to bugs. Polyspace flags the function-like macro definition.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 16-0-7

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator

## Description

### Rule Definition

*Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.*

### Rationale

If you attempt to use a macro identifier in a preprocessing directive, and you have not defined that identifier, then the preprocessor assumes that it has a value of zero. This value might not meet developer expectations.

### Polyspace Implementation

Polyspace flags an `#if` or `#elif` statement if it uses an undefined macro identifier.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Macro Identifiers

```
#if M == 0           //Noncompliant
#endif

#if defined (M)      //Complaint
#if M == 0           //Executes only when M is defined
#endif
#endif

#if defined (M) && (M == 0)  //Compliant
//...
#endif
```

This example shows various uses of `M` in preprocessing directives:

- The first `#if` clause uses the undefined identifier `M`. Because `M` is undefined when this preprocessor directive is evaluated, the compiler assumes that `M` is zero, which results in unexpected results. Such a use of undefined identifiers is not compliant with this rule. Polyspace flags the `#if` statement.
- The second and third `#if` statements use the undefined identifier `M` as the operand to the `defined` operator. These use of undefined identifiers are compliant with this rule.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 16-0-8

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token

## Description

### Rule Definition

*If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.*

### Rationale

The # character precedes a preprocessor directive when it is the first character on a line. If the # character is not immediately followed by a preprocessor directive, the preprocessor directive might be malformed.

Preprocessor directives might be used to exclude portions of code from compilation. The compiler excludes code until it encounters an #else, #elif, or #endif preprocessor directive. If one of those preprocessor directives is malformed, the compiler continues excluding code beyond the intended end point, resulting in bugs and unexpected behavior which can be difficult to diagnose.

Avoid malformed preprocessor directives by placing the preprocessor token directly after a # token. Specifically, do not place any characters other than white space between the # token and preprocessor token in #else and #endif directives.

### Polyspace Implementation

Polyspace raises this defect when the # character is followed by any character that is not part of a properly formed preprocessor token. A preprocessor token that is preceded or followed by any character other than white space causes Polyspace to raise this defect. Polyspace raises this defect when a preprocessor token is badly formed due to misspelling or improper capitalization.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Poorly Formed Preprocessor Tokens Following # Character

```
#define TESTING_H       //Compliant

namespace Example
{
#ifndef TESTING_H       //Compliant
    // code here
#elseX;                 //Noncompliant
    // code here
#else;                  //Compliant
```

```
    // code here
#endnif                 //Noncompliant
    // code here
  }

};
```

Because elseX is not a preprocessor directive and follows directly after the # character, Polyspace flags it as noncompliant.

#endnif is not a properly formed preprocessor directive. Polyspace flags it as noncompliant.

#define TESTING_H, #ifndef TESTING_H, and #else are properly formed preprocessor conditionals and are compliant with this rule.

### Check Information

**Group:** Preprocessing Directives
**Category:** Required

## Version History

**Introduced in R2013b**

### See Also

Check MISRA C++:2008 (-misra-cpp)

#### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 16-1-1

The defined preprocessor operator shall only be used in one of the two standard forms

## Description

### Rule Definition

*The defined preprocessor operator shall only be used in one of the two standard forms.*

### Rationale

The `defined` preprocessor operator checks whether an identifier is defined as a macro. In C, the only two permissible forms for this operator are:

- `defined (identifier)`
- `defined identifier`

Using any other form results in invalid code that compiler might not report. For instance, if you use expressions as arguments for the `defined` operator, the code is invalid. If the compiler does not report the invalid usage of `defined`, diagnosing the invalid code is difficult.

If your `#if` or similar preprocessor directives expand to create a `defined` statement, the code behavior is undefined. For instance:

```
#define DEFINED defined
#if DEFINED(X)
```

The `#if` preprocessor directive expands to form a `defined` operation. Depending on your environment, the code might behave in unexpected ways, leading to bugs that are difficult to diagnose.

To avoid invalid code, bugs, and undefined behavior, use only the permitted forms when using the `defined` operator.

### Polyspace Implementation

Polyspace flags incorrect usages of the `defined` operator, such as:

- The operator `defined` is used without an identifier.
- The operator `defined` appears after macro expansion.
- The operator `defined` is used with a complex expression.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use `defined` Operators With Identifiers

```
#if defined (X<Y)//Noncompliant
//...
#endif
#if defined (X) && defined (Y) &&(X<Y)//Compliant
//...
#endif
```

In this example, a block of code is conditionally executed only if the identifiers X and Y are defined and if X is smaller than Y. Constructing this condition by using an expression as the argument for the `defined` operator is not permissible and results in invalid code. Polyspace flags the impermissible `defined` statement. The permissible way to define such a condition is to use individual identifiers with `defined`.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 16-1-2

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related

## Description

### Rule Definition

*All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.*

### Rationale

You use preprocessor directives, such as `#if...#elif...#else...#endif`, to conditionally include or exclude blocks of code. If the different branches of such a directive reside in different source files, the code can be confusing. If all the branches are not included in a project, the code might behave in unexpected ways. To avoid confusion and unexpected behavior, keep the branches of a conditional preprocessor directive within the same source file.

### Polyspace Implementation

Polyspace raises a violation of this rule if either of these conditions are true:

- A corresponding `#if` directive cannot be found within a source file for every `#else`, `#elif`, or `#endif` directive.
- A corresponding `#endif` directive cannot be found within a source file for every `#if` directive.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Incomplete Conditional Preprocessor Directives

```
//file1.h
#if !defined (FILE)
//.....
#elif //Noncompliant
//...///
```

```
//file2.h
#else //Noncompliant
//...
#endif //Noncompliant
///
```

In this example, a conditional directive is split across two source files.

- In `file1.h`, the `#if` directive has no corresponding `#endif` directive. Polyspace flags the block.

- In `file2.h`, the `#else` and `#endif` directives have no corresponding `#if` directive. Polyspace flags both directives.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 16-2-1

The preprocessor shall only be used for file inclusion and include guards

## Description

### Rule Definition

*The preprocessor shall only be used for file inclusion and include guards.*

### Rationale

Aside from inclusion and include guarding, you might use preprocessor directives for other purposes such as defining constants or function-like macros. These preprocessor directives do not obey typical linkage and lack scoping mechanism or type safety. Preprocessor directives are less safe as compared to equivalent C++ features. For instance, a constant defined by using a `#define` statement retains its value across all scopes even if it is defined in a local scope. Using the `#define` preprocessor instead of a `constexpr` might lead to confusion if you define a constant differently in different scopes. Because a `constexpr` variable maintains a well defined scope, it is a safer alternative. The `constexpr` is efficient because it is a compile time constant.

Avoid preprocessor directives if they are not used for inclusion or include guards. Instead, use features such as inline functions, `const` or `constexpr` objects, and templates.

### Polyspace Implementation

Polyspace raises a violation of this rule in an included header file when either of these conditions is true:

- `#define` is used outside of an include guard. These `#define` statements typically define constants and function-like macros.
- `#ifndef` is used outside of an include guard.

Polyspace considers this idiom as the correct include guard idiom:

```
#ifndef <identifier>
#define <identifier>
#endif
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid #define Directives Outside Include Guards

| file1.h | main.cpp |
|---|---|
| `#ifndef MY_FILE`<br>`#define MY_FILE //Compliant`<br>`#endif`<br>`#define PI 3.1416 //Noncompliant`<br>`constexpr double pi = 3.1416;` | `#include"file1.h"`<br>`//...` |

In this example, include file `file1.h` contains two `#define` statements. The first `#define` directive is used within an include guard. This directive is compliant with this rule. The second `#define` directive defines the constant macro `PI`. This directive is noncompliant with this rule and Polyspace flags it. A better alternative is to use a `constexpr` variable.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 16-2-2

C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers

## Description

### Rule Definition

*C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.*

### Rationale

Aside from defining include guards, type qualifiers, and storage class specifiers, you might use C++ macros for other purposes such as defining constants or function-like macros. These macros do not obey typical linkage and lack scoping mechanism or type safety. Compared to available alternatives in C++, macros are less safe. For instance, a constant defined by using a `#define` statement retains its value across all scopes even if it is defined in a local scope. Using a macro instead of a `constexpr` might lead to confusion if you define a constant differently in different scopes. Because a `constexpr` variable maintains a well-defined scope, it is a safer alternative. The `constexpr` is efficient because it is a compile time constant.

Avoid macros if they are not used for defining include guards, type qualifiers, and storage class specifiers. Instead, use features such as inline function, `const` or `constexpr` objects, and function templates.

### Polyspace Implementation

The checker flags `#define` statements where the macros expand to something other than include guards, type qualifiers or storage class specifiers such as `static`, `inline`, `volatile`, `auto`, `register`, and `const`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Noncompliant Use of Macros

```
#ifndef IDENTIFIER //Compliant
#define IDENTIFIER //Compliant
#endif             //Compliant - Include guard

#define STOR extern      // Compliant - Storage class specifier
#define VOL volatile     //Compliant - Type qualifier

#define CLOCK (xtal/16)              // Noncompliant
#define PLUS2(X) ((X) + 2)           // Noncompliant
#define PI 3.14159F                  // Noncompliant
#define int32_t long                 // Noncompliant
#define STARTIF if(                  // Noncompliant
#define INIT(value) {(value), 0, 0}  // Noncompliant
```

```
#define HEADER "filename.h"          // Noncompliant
```

In this example, Polyspace flags all macros except those that define include guards, storage class specifiers, and type qualifiers.

### Check Information

**Group:** Preprocessing Directives
**Category:** Required

## Version History

**Introduced in R2013b**

### See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 16-2-3

Include guards shall be provided

## Description

### Rule Definition

*Include guards shall be provided.*

### Rationale

When a translation unit contains a complex hierarchy of nested header files, it is possible for a particular header file to be included more than once, leading to confusion. If this multiple inclusion produces multiple or conflicting definitions, then your program can have undefined or erroneous behavior.

For instance, suppose that a header file contains:

```
#ifdef _WIN64
   int env_var;
#elseif
   long int env_var;
#endif
```

If the header file is contained in two inclusion paths, one that defines the macro `_WIN64` and another that undefines it, you can have conflicting definitions of `env_var`.

To avoid multiple inclusion of the same file, add include guards to the beginning of header files. Use either of these formats:

-
  ```
  <start-of-file>
  // Comments allowed here
  #if !defined ( identifier )
  #define identifier
  // Contents of file
  #endif
  <end-of-file>
  ```

-
  ```
  <start-of-file>
  // Comments allowed here
  #ifndef identifier
  #define identifier
  // Contents of file
  #endif
  <end-of-file>
  ```

### Polyspace Implementation

The checker raises a violation if a header file does not contain an include guard.

For instance, this code uses an include guard for the `#define` and `#include` statements. This code does not violate the rule:

```
// Contents of a header file
#ifndef FILE_H
```

```
#define FILE_H
#include "libFile.h"
#endif
```

If you use include guards that do not adhere to the suggested format, Polyspace flags them. For instance:

- You might mistakenly use different identifiers in the `#ifndef` and `#define` statements:

  ```
  #ifndef MACRO
  #define MICRO
  //...
  #endif
  ```
- You might inadvertently use `#ifdef` instead of `#ifndef` or omit the `#define` statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Missing or Incorrectly Formatted Include Guard

| file1.h | file2.h | mainfile.cpp |
|---|---|---|
| `#ifndef MACRO`<br>`#define MICRO`<br>`//...`<br>`#endif` | `#ifdef DO_INCLUDE`<br>`#define DO_INCLUDE`<br>`void foo();`<br>`#endif` | `#include"file1.h"`<br>`#include"file2.h"`<br>`int main(){`<br>`    return 0;`<br>`}` |

In this example, two header files are included in the file `mainfile.cpp`.

- The include guard in `file1.h` queries the definition of `MACRO` but conditionally defines a different identifier `MICRO`, perhaps inadvertently. This include guard is incorrectly formatted. Polyspace flags the file.
- The include guard in `file2.h` uses `#ifdef` instead of `#ifndef`. This include guard is incorrect and Polyspace flags the file.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 16-2-4

The ', ", /* or // characters shall not occur in a header file name

## Description

### Rule Definition

*The ', ", /* or // characters shall not occur in a header file name.*

### Rationale

You include header files in a source file by enclosing its name between the delimiters <> or `""`. Using the characters `'`, ", `/*`, or `//` between the delimiters < and > is undefined behavior. Using the characters `'`, `/*`, or `//` between the " delimiters also results in undefined behavior. Depending on your environment and compiler, using these characters in a header filer name might result in unexpected behavior.

Avoid the characters `'`, ", `/*`, or `//` in a header file name.

### Polyspace Implementation

Polyspace raises a violation of this rule if the name of a header file includes one of these characters:

- `'`
- `"`
- `/*`
- `//`

If you use the character " between the delimiter ", Polyspace interprets the portion of the header name between two successive " delimiters as the header file name. For instance, in this code,

```
#include "foo\".h"
```

Polyspace interprets `foo\` as the header name. In such cases, Polyspace does not raise a violation of this rule, but because the compiler looks for a header file that does not exist, you might get a compilation warning.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Undefined Behavior

```
#include<dummy'file> //Noncompliant
#include<dummy"file> //Noncompliant

#include<dummy/*file> //Noncompliant
#include<dummy//file> //Noncompliant
```

```
#include "dummy'file" //Noncompliant
#include "dummy file"   //Compliant

#include "dummy/*file" //Noncompliant
#include "dummy//file" //Noncompliant
```

In this example, Polyspace flags the use of the characters `'`, `"`, `/*`, or `//` in header file names.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 16-2-5

The \ character should not occur in a header file name

## Description

### Rule Definition

*The \ character should not occur in a header file name.*

### Rationale

You include header files in a source file by enclosing its name between the delimiters <> or `""`. Using the character `\`between the delimiters < and > or between the delimiters `"` is undefined behavior. Using `/` in a header file name might result in unexpected behavior.

Some environments use the character `\` as a file name delimiter, for instance, when describing paths. Compilers for these environments might support the use of `\` in an `#include` directive.

### Polyspace Implementation

Polyspace raises a violation of this rule if the character `\` occurs in a header file name.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using \ in Header File Name

```
#include"incguard\5\.h" //Noncompliant
#include"inc\\guard.h" //Noncompliant
```

In this example, Polyspace flags the `#include` statements that have header file names where the `\` character occurs.

## Check Information

**Group:** Preprocessing Directives
**Category:** Advisory

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

**Topics**

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 16-2-6

The #include directive shall be followed by either a <filename> or "filename" sequence

## Description

### Rule Definition

*The #include directive shall be followed by either a <filename> or "filename" sequence.*

### Rationale

This rule applies only after macro replacement.

The code behavior is undefined if an `#include` directive does not use one of these forms:

- `#include <filename>`
- `#include "filename"`

Using other forms of `#include` directives is not permitted by ISO/IEC 14882:2003. To avoid unexpected behavior, avoid using malformed `#include` statements.

### Polyspace Implementation

Polyspace raises a violation of this rule if an `#include` directive is not followed by either `<filename>` or `"Filename"`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Malformed #include Directives

```
#include"incguard.h" //Compliant
#include<incguard.h> //Compliant

#define MH "myheader.h"
#define STR <string>
#include MH //Compliant
#include STR //Compliant

#include myfile.h //Noncompliant
```

In this example, Polyspace flags the directive that attempts to include `myfile.h` because it does not follow either of the permissible forms. The other directives follow the permissible forms after applicable macro replacements.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 16-3-1

There shall be at most one occurrence of the # or ## operators in a single macro definition

## Description

### Rule Definition

*There shall be at most one occurrence of the # or ## operators in a single macro definition.*

### Rationale

The evaluation of the # and ## preprocessor operators does not have a specified execution order. When more than one occurrence of the # or ## operators exists in a single macro definition, it is unclear which preprocessor operator is executed first by the compiler. The uncertainty of execution order might result in developer confusion or unexpected macro calculations. Use only one of the # or ## preprocessor operators for each macro definition.

### Polyspace Implementation

Polyspace raises this defect whenever more than one instance of the # or ## operators is used in a single macro definition.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Multiple # and ## Operators Used in a Single Macro Definition

```
#define STRING(X) { #X }                          //compliant
#define CONCAT(X, Y) {X ## Y}                     //compliant
#define STRING_CONCAT(x, y) {#x ## y}             //noncompliant
#define MULTI_CONCAT(x, y, xy, z) {x ## y ## z}   //noncompliant
```

Because the macro `STRING_CONCAT` uses both the # and ## operators, Polyspace flags the macro as noncompliant.

Polyspace flags the macro `MULTI_CONCAT` as noncompliant because it uses multiple ## operators.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 16-3-2

The # and ## operators should not be used

## Description

### Rule Definition

*The # and ## operators should not be used.*

### Rationale

The evaluation of the # and ## preprocessor operators does not have a specified execution order. Different compilers might evaluate these operators in different order of execution. The uncertainty of execution order might result in developer confusion or unexpected macro calculations. When possible, avoid using the # and ## preprocessor operators.

### Polyspace Implementation

Polyspace raises this advisory when the # or ## operators are used in a macro definition.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using # and ## Operators in Macro Definition

```
#define STRING(X) { #X }                            //noncompliant
#define CONCAT(X, Y) {X ## Y}                       //noncompliant
```

Because the macro `STRING(X)` uses the # operator, Polyspace flags the macro as noncompliant.

Polyspace flags the macro `CONCAT(X, Y)` as noncompliant because it uses the ## operator.

## Check Information

**Group:** Preprocessing Directives
**Category:** Advisory

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 16-6-1

All uses of the #pragma directive shall be documented

## Description

### Rule Definition

*All uses of the #pragma directive shall be documented.*

### Rationale

Because the behaviors of `#pragma` directives depend on the set of software, hardware, and compilers that you use, the developer's intent for a `#pragma` directive might be unclear. To clearly communicate the developer intent and the expected behavior of a `#pragma` directive, for each of these directives, document:

- The meaning
- The detailed expected behavior
- The implication of the directive for the code

Document the preceding factors in sufficient detail to show that you fully understand what the `#pragma` directives mean and how they might impact the code. Avoid using `#pragma` directives as much as possible. Encapsulate their use in dedicated functions whenever possible.

### Polyspace Implementation

To check this rule, list the pragmas that are allowed in source files by using the option `Allowed pragmas (-allowed-pragmas)`. If Polyspace finds a pragma not in the allowed pragma list, a violation is raised.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Preprocessing Directives
**Category:** Document

# Version History

**Introduced in R2016b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 17-0-1

Reserved identifiers, macros and functions in the Standard Library shall not be defined, redefined or undefined

## Description

### Rule Definition

*Reserved identifiers, macros and functions in the Standard Library shall not be defined, redefined or undefined.*

### Rationale

Redefining or undefining reserved identifiers, macros and functions from the Standard Library is not good practice. In some cases, these actions can lead to undefined behavior.

### Polyspace Implementation

The checker raises a violation if identifiers and macros from the Standard Library are defined, redefined or undefined.

In general, the checker considers identifiers and macros that begin with an underscore followed by an uppercase letter as reserved for the Standard Library.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Defining, Redefining, and Undefining Reserved C++ Identifiers, Macros, and Functions

```
#define __TIME__ 3  //Noncompliant
#undef __TIME__   //Noncompliant

#define __cplusplus 3 //Noncompliant
#undef __cplusplus //Noncompliant

#define break 3 //Noncompliant
#undef break //Noncompliant

#define pow 3 //Noncompliant
#undef pow //Noncompliant


#define example 3 //Compliant
#undef example //Compliant

#define example 7 //Compliant
```

In this example, Polyspace flags preprocessor directives that define or undefine reserved C++ identifiers and macros. For instance:

- Defining and undefining `__TIME__` is noncompliant because `__TIME__` is a reserved C++ macro.
- Defining and undefining `__cplusplus` is noncompliant because `__cplusplus` is a reserved C++ macro.
- Defining and undefining `break` is noncompliant because `break` is a reserved C++ identifier.
- Defining and undefining `pow` is noncompliant because `pow` is a reserved C++ function.
- Defining, undefining, and redefining `example` is compliant because `example` is not a reserved C++ macro, identifier, or function.

## Check Information

**Group:** Library Introduction
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 17-0-2

The names of standard library macros and objects shall not be reused

## Description

### Rule Definition

*The names of standard library macros and objects shall not be reused.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Library Introduction
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 17-0-3

The names of standard library functions shall not be overridden

## Description

### Rule Definition

*The names of standard library functions shall not be overridden.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Library Introduction
**Category:** Required

# Version History

**Introduced in R2018a**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 17-0-5

The setjmp macro and the longjmp function shall not be used

## Description

### Rule Definition

*The setjmp macro and the longjmp function shall not be used.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Library Introduction
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 18-0-1

The C library shall not be used

## Description

### Rule Definition

*The C library shall not be used.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Support Library
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 18-0-2

The library functions atof, atoi and atol from library <cstdlib> shall not be used

## Description

### Rule Definition

*The library functions atof, atoi and atol from library <cstdlib> shall not be used.*

### Rationale

Converting strings to a numeric value by using the functions `atof`, `atoi` and `atol` from the library <cstdlib> might result in error conditions. For instance, using the preceding functions might cause an error when the input string:

- Does not contain a number
- Contains a number, but is out of range
- Contains additional data after a number

When using the preceding functions, failure to convert a string to a numeric value might result in undefined behavior. To avoid undefined behavior and undetected errors, check the error state of output when converting strings to a numeric value.

### Polyspace Implementation

Polyspace flags the C standard library string-to-number functions of `atoi()`, `atol()`, and `atof()`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use C++ Library Functions for Converting Strings to Numeric Value

```
#include <cstdlib>
#include <iostream>
#include <string>

void foo() {
    std::string str1 = "7";
    std::string str2 = "3.1415";
    std::string str3 = "three";

    int myint1 = std::stoi(str1); //Compliant
    float myint2 = std::stof(str2); //Compliant
    long myint3 = std::stol(str3); //Compliant

    const char* str4 = "12";
    const char* str5 = "2.7182";
    const char* str6 = "undefinedError";
    int num4 = atoi(str4); //Noncompliant
    float num5 = atof(str5); //Noncompliant
    long num6 = atol(str6); //Noncompliant
    //...
}
```

In this example, Polyspace flags the use of C standard library functions for converting strings to numeric value. For instance:

- The string-to-number functions from the C standard library, such as `atoi()`, `atof()`, and `atol()` are noncompliant flagged because an invalid conversion results in undefined behavior.
- The string-to-number functions from the C++ standard library `std::stoi()`, `std::stof()`, and `std::stol()` are not flagged because an invalid conversion produces a `std::invalid_argument` exception, which is defined behavior.

### Check Information

**Group:** Language Support Library
**Category:** Required

## Version History

**Introduced in R2013b**

### See Also

`Check MISRA C++:2008 (-misra-cpp)`

#### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 18-0-3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used

## Description

### Rule Definition

*The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Support Library
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 18-0-4

The time handling functions of library <ctime> shall not be used

## Description

### Rule Definition

*The time handling functions of library <ctime> shall not be used.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Support Library
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 18-0-5

The unbounded functions of library <cstring> shall not be used

## Description

### Rule Definition

*The unbounded functions of library <cstring> shall not be used.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Support Library
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 18-2-1

The macro offsetof shall not be used

## Description

### Rule Definition

*The macro offsetof shall not be used.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Support Library
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 18-4-1

Dynamic heap memory allocation shall not be used

## Description

### Rule Definition

*Dynamic heap memory allocation shall not be used.*

### Rationale

Dynamic memory allocation uses heap memory, which can lead to issues such as memory leaks, data inconsistency, memory exhaustion, and nondeterministic behavior.

### Polyspace Implementation

The checker flags uses of the `malloc`, `calloc`, `realloc` and `free` functions, and non-placement versions of the `new` and `delete` operator.

The checker also flags uses of the `alloca` function. Though memory leak cannot happen with the `alloca` function, other issues associated with dynamic memory allocation can still occur.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Support Library
**Category:** Required

## Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 18-7-1

The signal handling facilities of <csignal> shall not be used

## Description

### Rule Definition

*The signal handling facilities of <csignal> shall not be used.*

### Rationale

Signal handling functions such as `signal` contains undefined and implementation-specific behavior.

You have to be very careful when using `signal` to avoid these behaviors.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Support Library
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Return from computational exception signal handler` | `Signal call in multithreaded program` | `Shared data access within signal handler` | `Function called from signal handler not asynchronous-safe` | `Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

# MISRA C++:2008 Rule 19-3-1

The error indicator errno shall not be used

## Description

### Rule Definition

*The error indicator errno shall not be used.*

### Rationale

Observing this rule encourages the good practice of not relying on `errno` to check error conditions.

Checking `errno` is not sufficient to guarantee absence of errors. Functions such as `fopen` might not set `errno` on error conditions. Often, you have to check the return value of such functions for error conditions.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `errno`

```
#include <cstdlib>
#include <cerrno>

void func (const char* str) {
    errno = 0;  // Noncompliant
    int i = atoi(str);
    if(errno != 0) { // Noncompliant
        //Handle Error
    }
}
```

The use of `errno` violates this rule. The function `atoi` is not required to set `errno` if the input string cannot be converted to an integer. Checking `errno` later does not safeguard against possible failures in conversion.

## Check Information

**Group:** Diagnostic Library
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Misuse of errno` | `Misuse of errno in a signal handler` | `Check MISRA C++:2008 (-misra-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

# MISRA C++:2008 Rule 27-0-1

The stream input/output library <cstdio> shall not be used

## Description

### Rule Definition

*The stream input/output library <cstdio> shall not be used.*

### Rationale

Functions in `cstdio` such as `gets`, `fgetpos`, `fopen`, `ftell`, etc. have unspecified, undefined and implementation-defined behavior.

For instance:

- The `gets` function:

  ```
  char * gets ( char * buf );
  ```

  does not check if the number of characters provided at the standard input exceeds the buffer `buf`. The function can have unexpected behavior when the input exceeds the buffer.
- The `fopen` function has implementation-specific behavior related to whether it sets `errno` on errors or whether it accepts additional characters following the standard mode specifiers.

### Polyspace Implementation

Polyspace reports a violation of this rule if you use the functions from the `cstdio` library. Polyspace detects the use of these `cstdio` functions:

- File operation functions such as `remove` and `rename`.
- File access functions such as `fclose`,`fflush`, and `fopen`.
- Formatted input/output functions such as `fprintf`, `fscanf`, `printf`, and `scanf`.
- Character input output functions such as `fgetc`, `fgets`, `fputc`, and `getc`.
- Direct input/output functions such as `fread` and `fwrite`.
- File positioning functions such as `fgetpos` and `fsetpos`.
- Error handling functions such as `clearerr`, `ferror`, and `perror`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `gets`

```
#include <cstdio>
```

```
void func()
{
    char array[10];
    fgets(array, sizeof array, stdin); //Noncompliant
}
```

The use of `fgets` violates this rule.

## Check Information

**Group:** Input/output Library<br>
**Category:** Required

# Version History

**Introduced in R2013b**

## See Also

`Check MISRA C++:2008 (-misra-cpp)`

### Topics

“Check for and Review Coding Standard Violations”

25

# CERT C++ Rules

# Acknowledgement

This software has been created by MathWorks incorporating portions of: the “SEI CERT-C Website,” © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, ”SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition,” © 2016 Carnegie Mellon University, and “SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition” © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.

ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.

# CERT C++: DCL30-C

Declare objects with appropriate storage durations

## Description

### Rule Definition

*Declare objects with appropriate storage durations.*

### Polyspace Implementation

The rule checker checks for **Pointer or reference to stack variable leaving scope**.

## Examples

### Pointer or reference to stack variable leaving scope

#### Issue

**Pointer or reference to stack variable leaving scope** occurs when a pointer or reference to a local variable leaves the scope of the variable. For instance:

- A function returns a pointer to a local variable.
- A function performs the assignment `globPtr = &locVar`. `globPtr` is a global pointer variable and `locVar` is a local variable.
- A function performs the assignment `*paramPtr = &locVar`. `paramPtr` is a function parameter that is, for instance, an `int**` pointer and `locVar` is a local `int` variable.
- A C++ method performs the assignment `memPtr = &locVar`. `memPtr` is a pointer data member of the class the method belongs to. `locVar` is a variable local to the method.

The defect also applies to memory allocated using the `alloca` function. The defect does not apply to static, local variables. Polyspace assumes that the local objects within a function definition are in the same scope.

#### Risk

Local variables are allocated an address on the stack. Once the scope of a local variable ends, this address is available for reuse. Using this address to access the local variable value outside the variable scope can cause unexpected behavior.

If a pointer to a local variable leaves the scope of the variable, Polyspace Bug Finder highlights the defect. The defect appears even if you do not use the address stored in the pointer. For maintainable code, it is a good practice to not allow the pointer to leave the variable scope. Even if you do not use the address in the pointer now, someone else using your function can use the address, causing undefined behavior.

#### Fix

Do not allow a pointer or reference to a local variable to leave the variable scope.

#### Example - Pointer to Local Variable Returned from Function

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0; //Noncompliant
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In this example, `func1` returns a pointer to local variable `ret`.

In `main`, `ptr` points to the address of the local variable. When `ptr` is accessed in `func2`, the access is illegal because the scope of `ret` is limited to `func1`,

#### Example - Pointer to Local Variable Escapes Through Lambda Expression

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;  //Noncompliant
  auto adder = [&] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

In this example, the `createAdder` function defines a lambda expression `adder` that captures the local variable `addThis` by reference. The scope of `addThis` is limited to the `createAdder` function. When the object returned by `createAdder` is called, a reference to the variable `addThis` is accessed outside its scope. When accessed in this way, the value of `addThis` is undefined.

#### Correction – Capture Local Variables by Copy in Lambda Expression Instead of Reference

If a function returns a lambda expression object, avoid capturing local variables by reference in the lambda object. Capture the variables by copy instead.

Variables captured by copy have the same lifetime as the lambda object, but variables captured by reference often have a smaller lifetime than the lambda object itself. When the lambda object is used, these variables accessed outside scope have undefined values.

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [=] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

## Check Information

**Group:** 01. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL30-C

# CERT C++: DCL39-C

Avoid information leakage in structure padding

## Description

### Rule Definition

*Avoid information leakage in structure padding.*

### Polyspace Implementation

The rule checker checks for **Information leak via structure padding**.

## Examples

### Information leak via structure padding

#### Issue

**Information leak via structure padding** occurs when you do not initialize the padding data of a structure or union before passing it across a trust boundary. A compiler adds padding bytes to the structure or union to ensure a proper memory alignment of its members. The bit-fields of the storage units can also have padding bits.

**Information leak via structure padding** raises a defect when:

- You call an untrusted function with structure or union pointer type argument containing uninitialized padding data.

  All external functions are considered untrusted.
- You copy or assign a structure or union containing uninitialized padding data to an untrusted object.

  All external structure or union objects, the output parameters of all externally linked functions, and the return pointer of all external functions are considered untrusted objects.

#### Risk

The padding bytes of the passed structure or union might contain sensitive information that an untrusted source can access.

#### Fix

- Prevent the addition of padding bytes for memory alignment by using the `pack` pragma or attribute supported by your compiler.
- Explicitly declare and initialize padding bytes as fields within the structure or union.
- Explicitly declare and initialize bit-fields corresponding to padding bits, even if you use the `pack` pragma or attribute supported by your compiler.

#### Example - Structure with Padding Bytes Passed to External Function

```
#include <stddef.h>
#include <stdlib.h>
```

```
#include <string.h>

typedef struct s_padding
{
 /* Padding bytes may be introduced between
 * 'char c' and 'int i'
 */
        char c;
    int i;

/*Padding bits may be introduced around the bit-fields
* even if you use "#pragma pack" (Windows) or
* __attribute__((__packed__)) (GNU)*/

    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;



/* External function */
extern void copy_object(void *out, void *in, size_t s);

void func(void *out_buffer)
{
/*Padding bytes not initialized*/

    S_Padding s = {'A', 10, 1, 3, {}};
/*Structure passed to external function*/

    copy_object((void *)out_buffer, (void *)&s, sizeof(s)); //Noncompliant
}

void main(void)
{
    S_Padding s1;
    func(&s1);
}
```

In this example, structure `s1` can have padding bytes between the `char c` and `int i` members. The bit-fields of the storage units of the structure can also contain padding bits. The content of the padding bytes and bits is accessible to an untrusted source when `s1` is passed to `func`.

**Correction — Use pack Pragma to Prevent Padding Bytes**

One possible correction in Microsoft Visual Studio is to use `#pragma pack()` to prevent padding bytes between the structure members. To prevent padding bits in the bit-fields of `s1`, explicitly declare and initialize the bit-fields even if you use `#pragma pack()`.

```
 #include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define CHAR_BIT 8

#pragma pack(push, 1)
```

```
typedef struct s_padding
{
/*No Padding bytes when you use "#pragma pack" (Windows) or
* __attribute__((__packed__)) (GNU)*/
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
/* Padding bits explicitely declared */
    unsigned int bf_filler : sizeof(unsigned) * CHAR_BIT - 3;
    unsigned char buffer[20];
}

    S_Padding;

#pragma pack(pop)


/* External function */
extern void copy_object(void *out, void *in, size_t s);



void func(void *out_buffer)
{
    S_Padding s = {'A', 10, 1, 3, 0 /* padding bits */, {}};
    copy_object((void *)out_buffer, (void *)&s, sizeof(s));
}

void main(void)
{
    S_Padding s1;
    func(&s1);
}
```

## Check Information

**Group:** 01. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

DCL39-C

# CERT C++: DCL40-C

Do not create incompatible declarations of the same function or object

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.

### Rule Definition

*Do not create incompatible declarations of the same function or object.*

### Polyspace Implementation

The rule checker checks for **Declaration mismatch**.

## Examples

### Declaration mismatch

#### Issue

**Declaration mismatch** occurs when a function or variable declaration does not match other instances of the function or variable.

#### Risk

When a mismatch occurs between two variable declarations in different compilation units, a typical linker follows an algorithm to pick one declaration for the variable. If you expect a variable declaration that is different from the one chosen by the linker, you can see unexpected results when the variable is used.

A similar issue can occur with mismatch in function declarations.

#### Fix

The fix depends on the type of declaration mismatch. If both declarations indeed refer to the same object, use the same declaration. If the declarations refer to different objects, change the names of the one of the variables. If you change a variable name, remember to make the change in all places that use the variable.

Sometimes, declaration mismatches can occur because the declarations are affected by previous preprocessing directives. For instance, a declaration occurs in a macro, and the macro is defined on one inclusion path but undefined in another. These declaration mismatches can be tricky to debug. Identify the divergence between the two inclusion paths and fix the conflicting macro definitions.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.

- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Inconsistent Declarations in Two Files

*file1.c*

```
int foo(void) {
    return 1;
}
```

*file2.c*

```
double foo(void); //Noncompliant

int bar(void) {
    return (int)foo();
}
```

In this example, *file1.cpp* declares `foo()` as returning an integer. In *file2.cpp*, `foo()` is declared as returning a double. This difference raises a defect on the second instance of `foo` in *file2*.

#### Correction — Align the Function Return Values

One possible correction is to change the function declarations so that they match. In this example, by changing the declaration of `foo` in *file2.cpp* to match *file1.cpp*, the defect is fixed.

*file1.c*

```
int foo(void) {
    return 1;
}
```

*file2.c*

```
int foo(void);

int bar(void) {
    return foo();
}
```

#### Example - Inconsistent Structure Alignment

*test1.c*

```
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

*test2.c*

```
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

*circle.h*

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;
```

*square.h*

```
extern struct aSquare {
    unsigned int side:1;
} square; //Noncompliant
```

In this example, a declaration mismatch defect is raised on `square` in *square.h* because Polyspace infers that `square` in *square.h* does not have the same alignment as `square` in *test2.cpp*. This error occurs because the `#pragma pack(1)` statement in *circle.h* declares specific alignment. In *test2.cpp*, *circle.h* is included before *square.h*. Therefore, the `#pragma pack(1)` statement from *circle.h* is not reset to the default alignment after the `aCircle` structure. Because of this omission, *test2.cpp* infers that the `aSquare square` structure also has an alignment of `1` byte.

#### Correction — Close Packing Statements

One possible correction is to reset the structure alignment after the `aCircle` struct declaration. For the GNU or Microsoft Visual compilers, fix the defect by adding a `#pragma pack()` statement at the end of *circle.h*.

*test1.c*

```
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

*test2.c*

```
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

*circle.h*

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;

#pragma pack()
```

*square.h*

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

Other compilers require different `#pragma pack` syntax. For your syntax, see the documentation for your compiler.

#### Correction — Use the `Ignore pragma pack directives` Option

One possible correction is to add the `Ignore pragma pack directives` option to your Bug Finder analysis. If you want the structure alignment to change for each structure, and you do not want to see this **Declaration mismatch** defect, use this correction.

1. On the Configuration pane, select the **Advanced Settings** pane.
2. In the **Other** box, enter `-ignore-pragma-pack`.
3. Rerun your analysis.

The **Declaration mismatch** defect is resolved.

## Check Information

**Group:** 01. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

DCL40-C

# CERT C++: DCL50-CPP

Do not define a C-style variadic function

## Description

### Rule Definition

*Do not define a C-style variadic function.*

### Polyspace Implementation

The rule checker checks for **Function definition with ellipsis notation**.

## Examples

### Function definition with ellipsis notation

#### Issue

The issue occurs when you define a function using the ellipsis notation.

```
int func( const char* format, ...);
```

## Check Information

**Group:** 01. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

DCL50-CPP

# CERT C++: DCL51-CPP

Do not declare or define a reserved identifier

## Description

### Rule Definition

*Do not declare or define a reserved identifier.*

### Polyspace Implementation

This checker checks for:

- **Defining or undefining reserved identifier or macro**
- **User-defined literal operator not starting with underscore**

## Examples

### Defining or undefining reserved identifier or macro

#### Issue

The issue occurs when you define, redefine, or undefine a reserved identifier, macro, or function in the Standard Library.

In general, the checker considers identifiers and macros that begin with an underscore followed by an uppercase letter as reserved for the Standard Library.

### User-defined literal operator not starting with underscore

#### Issue

This issue occurs when you define operators of the form

`operator "" *suffix*`

where *`suffix`* does not begin with an underscore or following the underscore, contains characters other than letters (numbers, special characters, and so on).

#### Risk

Since C++11, you can add suffixes to literals that convert numeric values under the hood. For instance, in code where you perform all calculations in a common unit, you can leave unit conversions to dedicated operators and simply use literal suffixes for the units when defining constant values.

In this example, the literal suffixes `_m` and `_km` resolve to calls to `operator"" _m()` and `operator"" _km()` respectively. The operators ensure that all values are converted to the same unit.

```
constexpr long double operator"" _m(long double metres) {
    return metres;
}
```

```
constexpr long double operator"" _km(long double kilometres) {
    return 1000*kilometres;
}
...
long double minSteps = 100.0_m;
long double interCityDist = 100.0_km;
```

User defined literal suffixes must begin with an underscore (_). Literal suffixes not beginning with underscore are reserved for Standard Library.

#### Fix

Make sure that user-defined literal operators begin with an underscore followed by letters only.

## Check Information

**Group:** 01. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

DCL51-CPP

# CERT C++: DCL52-CPP

Never qualify a reference type with const or volatile

## Description

### Rule Definition

*Never qualify a reference type with const or volatile.*

### Polyspace Implementation

The rule checker checks for these issues:

- **C++ reference type qualified with const or volatile**.
- **C++ reference to const-qualified type with subsequent modification**.

## Examples

### C++ reference type qualified with const or volatile

#### Issue

**const-Qualified Reference Type** occurs when a variable with reference type is declared with the `const` or `volatile` qualifier, for instance:

```
char &const c;
```

#### Risk

The C++14 Standard states that `const` or `volatile` qualified references are ill formed (unless they are introduced through a `typedef`, in which case they are ignored). For instance, a reference to one variable cannot be made to refer to another variable. Therefore, using the `const` qualifier is not required for a variable with a reference type.

Often the use of these qualifiers indicate a coding error. For instance, you meant to declare a reference to a `const`-qualified type:

```
char const &c;
```

but instead declared a `const`-qualified reference:

```
char &const c;
```

If your compiler does not detect the error, you can see unexpected results. For instance, you might expect `c` to be immutable but see a different value of `c` compared to its value at declaration.

#### Fix

See if the `const` or `volatile` qualifier is incorrectly placed. For instance, see if you wanted to refer to a `const`-qualified type and entered:

```
char &const c;
```

instead of:

```
char const &c;
```

If the qualifier is incorrectly placed, fix the error. Place the `const` or `volatile`qualifier before the & operator. Otherwise, remove the redundant qualifier.

#### Example – `const`-Qualified Reference Type

```
int func (int &const iRef) { //Noncompliant
    iRef++;
    return iRef%2;
}
```

In this example, `iRef` is a `const`-qualified reference type. Since `iRef` cannot refer to another variable, the `const` qualifier is redundant.

#### Correction — Remove `const` Qualifier

Remove the redundant `const` qualifier. Since `iRef` is modified in `func`, it is not meant to refer to a `const`-qualified variable. Moving the `const` qualifier before & will cause a compilation error.

```
int func (int &iRef) {
    iRef++;
    return iRef%2;
}
```

#### Correction — Fix Placement of `const` Qualifier

If you do not identify to modify `iRef` in `func`, declare `iRef` as a reference to a `const`-qualified variable. Place the `const` qualifier before the & operator. Make sure you do not modify `iRef` in `func`.

```
int func (int const &iRef) {
    return (iRef+1)%2;
}
```

### C++ reference to const-qualified type with subsequent modification

#### Issue

This defect occurs when a variable that refers to a `const`-qualified type is modified after declaration.

For instance, in this example, `refVal` has a type `const int &`, but its value is modified in a subsequent statement.

```
using constIntRefType = const int &;
void func(constIntRefType refVal, int val){
   ...
   refVal = val; //refVal is modified
   ...
}
```

#### Risk

The `const` qualifier on a reference type implies that a variable of the type is initialized at declaration and will not be subsequently modified.

Compilers can detect modification of references to `const`-qualified types as a compilation error. If the compiler does not detect the error, the behavior is undefined. Polyspace flags this defect regardless of a compilation error.

#### Fix

Avoid modification of `const`-qualified reference types. If the modification is required, remove the `const` qualifier from the reference type declaration.

#### Example – Modification of const-qualified Reference Types

```
typedef const int cint;
typedef cint& ref_to_cint;

void func(ref_to_cint refVal, int initVal){ //Noncompliant
   refVal = initVal;
}
```

In this example, `ref_to_cint` is a reference to a `const`-qualified type. The variable `refVal` of type `ref_to_cint` is supposed to be initialized when `func` is called and not modified subsequently. The modification violates the contract implied by the `const` qualifier. Because `refVal` is a `const` reference, the compilation might fail. Polyspace flags the violation.

#### Correction — Avoid Modification of const-qualified Reference Types

One possible correction is to avoid the `const` in the declaration of the reference type.

```
typedef int& ref_to_int;

void func(ref_to_int refVal, int initVal){
   refVal = initVal;
}
```

## Check Information

**Group:** 01. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL52-CPP

# CERT C++: DCL53-CPP

Do not write syntactically ambiguous declarations

## Description

### Rule Definition

*Do not write syntactically ambiguous declarations.*

### Polyspace Implementation

The rule checker checks for **Ambiguous Declaration Syntax**.

## Examples

### Ambiguous declaration syntax

#### Issue

This defect occurs when it is not clear from a declaration whether an object declaration or function/parameter declaration is intended. The ambiguity is often referred to as most vexing parse.

For instance, these declarations are ambiguous:

- `ResourceType aResource();`

  It is not immediately clear if `aResource` is a function returning a variable of type `ResourceType` or an object of type `ResourceType`.
- `TimeKeeper aTimeKeeper(Timer());`

  It is not immediately clear if `aTimeKeeper` is an object constructed with an unnamed object of type `Timer` or a function with an unnamed function pointer type as parameter. The function pointer refers to a function with no argument and return type `Timer`.

The checker does not flag ambiguous declarations with global scope. For instance, the analysis does not flag declarations with global scope using the format *`Type`* `a()` where *`Type`* is a class type with a default constructor. The analysis interprets `a` as a function returning the type *`Type`*.

#### Risk

In case of an ambiguous declaration, the C++ Standard chooses a specific interpretation of the syntax. For instance:

- `ResourceType aResource();`

  is interpreted as a declaration of a function `aResource`.
- `TimeKeeper aTimeKeeper(Timer());`

  is interpreted as a declaration of a function `aTimeKeeper` with an unnamed parameter of function pointer type.

If you or another developer or code reviewer expects a different interpretation, the results can be unexpected.

For instance, later you might face a compilation error that is difficult to understand. Since the default interpretation indicates a function declaration, if you use the function as an object, compilers might report a compilation error. The compilation error indicates that a conversion from a function to an object is being attempted without a suitable constructor.

**Fix**

Make the declaration unambiguous. For instance, fix these ambiguous declarations as follows:

- `ResourceType aResource();`

  *Object declaration*:

  If the declaration refers to an object initialized with the default constructor, rewrite it as:

  ```
  ResourceType aResource;
  ```

  prior to C++11, or as:

  ```
  ResourceType aResource{};
  ```

  after C++11.

  *Function declaration*:

  If the declaration refers to a function, use a typedef for the function.

  ```
  typedef ResourceType(*resourceFunctionType)();
  resourceFunctionType aResource;
  ```
- `TimeKeeper aTimeKeeper(Timer());`

  *Object declaration*:

  If the declaration refers to an object `aTimeKeeper` initialized with an unnamed object of class `Timer`, add an extra pair of parenthesis:

  ```
  TimeKeeper aTimeKeeper( (Timer()) );
  ```

  prior to C++11, or use braces:

  ```
  TimeKeeper aTimeKeeper{Timer{}};
  ```

  after C++11.

  *Function declaration*:

  If the declaration refers to a function `aTimeKeeper` with a unnamed parameter of function pointer type, use a named parameter instead.

  ```
  typedef Timer(*timerType)();
  TimeKeeper aTimeKeeper(timerType aTimer);
  ```

**Example – Function or Object Declaration**

```
class ResourceType {
      int aMember;
    public:
      int getMember();
};
```

```
void getResource() {
    ResourceType aResource(); //Noncompliant
}
```

In this example, `aResource` might be used as an object but the declaration syntax indicates a function declaration.

#### Correction — Use {} for Object Declaration

One possible correction (after C++11) is to use braces for object declaration.

```
class ResourceType {
      int aMember;
    public:
      int getMember();
};

void getResource() {
    ResourceType aResource{};
}
```

#### Example – Unnamed Object or Unnamed Function Parameter Declaration

```
class MemberType {};

class ResourceType {
      MemberType aMember;
    public:
      ResourceType(MemberType m) {aMember = m;}
      int getMember();
};

void getResource() {
    ResourceType aResource(MemberType());  //Noncompliant
}
```

In this example, `aResource` might be used as an object initialized with an unnamed object of type `MemberType` but the declaration syntax indicates a function with an unnamed parameter of function pointer type. The function pointer points to a function with no arguments and type `MemberType`.

#### Correction — Use {} for Object Declaration

One possible correction (after C++11) is to use braces for object declaration.

```
class MemberType {};

class ResourceType {
      MemberType aMember;
    public:
      ResourceType(MemberType m) {aMember = m;}
      int getMember();
};

void getResource() {
    ResourceType aResource{MemberType()};
}
```

#### Example – Unnamed Object or Named Function Parameter Declaration

```
class Integer {
    int aMember;
public:
    Integer(int d) {aMember = d;}
    int getMember();
};

int aInt = 0;
void foo(){
    Integer aInteger(Integer(aInt)); //Noncompliant
}
```

In this example, `aInteger` might be an object constructed with an unnamed object `Integer(aInt)` (an object of class `Integer` which itself is constructed using the variable `aInt`). However, the declaration syntax indicates that `aInteger` is a function with a named parameter `aInt` of type `Integer` (the superfluous parenthesis is ignored).

#### Correction — Use of {} for Object Declaration

One possible correction (after C++11) is to use {} for object declaration.

```
class Integer {
    int aMember;
public:
    Integer(int d) {aMember = d;}
    int getMember();
};

int aInt = 0;
void foo(){
    Integer aInteger(Integer{aInt});
}
```

#### Correction — Remove Superfluous Parenthesis for Named Parameter Declaration

If `aInteger` is a function with a named parameter `aInt`, remove the superfluous () around `aInt`.

```
class Integer {
       int aMember;
    public:
       Integer(int d) {aMember = d;}
       int getMember();
};

Integer aInteger(Integer aInt);
```

## Check Information

**Group:** 01. Declarations and Initialization (DCL)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL53-CPP

# CERT C++: DCL54-CPP

Overload allocation and deallocation functions as a pair in the same scope

## Description

### Rule Definition

*Overload allocation and deallocation functions as a pair in the same scope.*

### Polyspace Implementation

The rule checker checks for **Missing overload of allocation or deallocation function**.

## Examples

### Missing overload of allocation or deallocation function

#### Issue

**Missing overload of allocation or deallocation function** occurs when you overload `operator new` but do not overload the corresponding `operator delete`, or vice versa.

#### Risk

You typically overload `operator new` to perform some bookkeeping in addition to allocating memory on the free store. Unless you overload the corresponding `operator delete`, it is likely that you omitted some corresponding bookkeeping when deallocating the memory.

The defect can also indicate a coding error. For instance, you overloaded the placement form of `operator new[]`:

```
void *operator new[](std::size_t count, void *ptr);
```

but the non-placement form of `operator delete[]`:

```
void operator delete[](void *ptr);
```

instead of the placement form:

```
void operator delete[](void *ptr, void *p );
```

#### Fix

When overloading `operator new`, make sure that you overload the corresponding `operator delete` in the same scope, and vice versa.

For instance, in a class, if you overload the placement form of `operator new`:

```
class MyClass {
   void* operator new  ( std::size_t count, void* ptr ){
   //...
   }
};
```

Make sure that you also overload the placement form of `operator delete`:

```
class MyClass {
   void operator delete  ( void* ptr, void* place ){
   ...
   }
};
```

To find the `operator delete` corresponding to an `operator new`, see the reference pages for `operator new` and `operator delete`.

**Example – Mismatch Between Overloaded `operator new` and `operator delete`**

```
#include <new>
#include <cstdlib>

int global_store;

void update_bookkeeping(void *allocated_ptr, bool alloc) {
   if(alloc)
      global_store++;
   else
      global_store--;
}


void *operator new(std::size_t size, const std::nothrow_t& tag);
void *operator new(std::size_t size, const std::nothrow_t& tag) { //Noncompliant
    void *ptr = (void*)malloc(size);
    if (ptr != nullptr)
        update_bookkeeping(ptr, true);
    return ptr;
}

void operator delete[](void *ptr, const std::nothrow_t& tag);
void operator delete[](void* ptr, const std::nothrow_t& tag) { //Noncompliant
    update_bookkeeping(ptr, false);
    free(ptr);
}
```

In this example, the operators `operator new` and `operator delete[]` are overloaded but there are no overloads of the corresponding `operator delete` and `operator new[]` operators.

The overload of `operator new` calls a function `update_bookkeeping` to change the value of a global variable `global_store`. If the default `operator delete` is called, this global variable is unaffected, which might defy developer's expectations.

**Correction – Overload the Correct Form of `operator delete`**

If you want to overload `operator new`, overload the corresponding form of `operator delete` in the same scope.

```
#include <new>
#include <cstdlib>

int global_store;

void update_bookkeeping(void *allocated_ptr, bool alloc) {
   if(alloc)
      global_store++;
   else
      global_store--;
}


void *operator new(std::size_t size, const std::nothrow_t& tag);
void *operator new(std::size_t size, const std::nothrow_t& tag) {
    void *ptr = (void*)malloc(size);
    if (ptr != nullptr)
        update_bookkeeping(ptr, true);
    return ptr;
}

void operator delete(void *ptr, const std::nothrow_t& tag);
void operator delete(void* ptr, const std::nothrow_t& tag) {
    update_bookkeeping(ptr, false);
    free(ptr);
}
```

## Check Information

**Group:** 01. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL54-CPP

# CERT C++: DCL55-CPP

Avoid information leakage when passing a class object across a trust boundary

## Description

### Rule Definition

*Avoid information leakage when passing a class object across a trust boundary.*[7]

### Polyspace Implementation

The rule checker checks for **Information leakage due to structure padding**.

## Examples

### Information Leakage Due to Structure Padding

#### Issue

**Information leakage due to structure padding** occurs when sending a pointer to a data structure across a trust boundary without first initializing any padding bits in the data structure.

#### Risk

Class objects can contain padding bits. Because padding bits are defined by implementation, the layout of a class object can be unique between different compilers and architectures. Padding bits can be present at any location within a class object instance.

Sensitive information such as personal data, passwords, or pointers to kernel data structures can remain within the padding bits if the data is not properly initialized. If a class object is sent across a trust barrier, the sensitive data stored within padding bits can be exposed.

#### Fix

Verify that no sensitive data is contained within the padding bits before sending a class object instance across a trust boundary. Explicitly declare padding bits as fields within the class object or serialize the data before external use.

[7] *This software has been created by MathWorks incorporating portions of: the "SEI CERT-C Website," © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, "SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition," © 2016 Carnegie Mellon University, and "SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition" © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

### Example – Possible Leak of Sensitive Data Via Structure Padding

In this example, the structure `test` contains padding bits in order to properly align class data members. These padding bits may contain sensitive data which could be leaked when the data is copied to user space.

```
#include <cstddef>

struct example {
    int car;
    char model;
    int vin;
};

extern int send_to_external(void* dest, void* src, std::size_t size);

void processData(void* user_buffer) {
    example test{};
    test.car = 1;
    test.model = 2;
    test.vin = 3;

    send_to_external(user_buffer, &test, sizeof(test));//Noncompliant
}
```

### Correction – Serialize Data Structure Before Copy

Serialize the data structure before you copy it to an untrusted context so no padding bits contain sensitive data.

```
#include <cstddef>
#include <cstring>

struct example {
    int car;
    char model;
    int vin;
};

extern int send_to_external(void* dest, void* src, std::size_t size);

void processData(void* user_buffer) {
    example test{ 1, 2, 3 };

    unsigned char buff[sizeof(test)];
    std::size_t offset = 0;

    std::memcpy(buff + offset, &test.car, sizeof(test.car));
    offset += sizeof(test.car);
    std::memcpy(buff + offset, &test.model, sizeof(test.model));
    offset += sizeof(test.model);
    std::memcpy(buff + offset, &test.vin, sizeof(test.vin));
    offset += sizeof(test.vin);

    send_to_external(user_buffer, buff, offset /* size of info copied */);
}
```

## Check Information

**Group:** Rule 01. Declarations and Initialization (DCL)

# Version History

**Introduced in R2022b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL55-CPP

# CERT C++: DCL56-CPP

Avoid cycles during initialization of static objects

## Description

### Rule Definition

*Avoid cycles during initialization of static objects.*[8]

### Polyspace Implementation

The rule checker checks for these issues:

- **Recursive initialization of static variables**
- **Undetermined initialization order of global variables**

## Examples

### Recursive Initialization of Static Variables

#### Issue

This issue occurs when you declare a static variable in a function and initialize the static variable by calling the same function.

```
int foo(int) {
    static int var = foo(0);
    //...
}
```

#### Risk

Recursively initializing a static variable might result in undefined behavior. The code does not need to be infinitely recursive for undefined behavior to occur.

#### Fix

Avoid recursive calls when initializing variables. For example, use zero-initialization at the declaration of a static variable, then compute the values.

[8] *This software has been created by MathWorks incorporating portions of: the “SEI CERT-C Website,” © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, ”SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition,” © 2016 Carnegie Mellon University, and “SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition” © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

### Example

In the following example, the behavior is undefined because the initialization of the static variable `buffer` is within the `fibonacci` function, and the initialization of `buffer` requires reentering the same function `fibonacci`.

```
#include <stdexcept>

int fibonacci(int i) noexcept(false)
{
    if (i < 0) {
        //no negative fibonacci number
        throw std::domain_error("Input must be nonegative");
    }
    static const int buffer[] = { fibonacci(0), fibonacci(1),  //Noncompliant
        fibonacci(2), fibonacci(3), fibonacci(4), fibonacci(5),
        fibonacci(6), fibonacci(7) };

    if (i < (sizeof(buffer) / sizeof(int))) {
        return buffer[i];
    }

    return i > 0 ? fibonacci(i - 1) + fibonacci(i - 2) : 1;
}
```

The `fibonacci` function cannot calculate the next value until computation of the previous value is complete.

### Correction

Use zero-initialization to determine if every member of the array an assigned value. If not, recursively compute the value.

```
#include <stdexcept>

int fibonacci(int i) noexcept(false)
{
    if (i < 0) {
        //no negative fibonacci number
        throw std::domain_error("Input must be nonegative");
    }

    static int buffer[7]; // default zero-initailization

    if (i < sizeof(buffer) / sizeof(int))
    {
        if (0 == buffer[i])
        {
            buffer[i] = i > 0 ? fibonacci(i - 1) + fibonacci(i - 2) : 1;
        }
        return buffer[i];
    }

}
```

### Undetermined Initialization Order of Global Variables

#### Issue

This issue occurs when a global variable initialization in a file depends on global variables that are initialized in another file.

#### Risk

If a static variable is used before it is initialized, it may cause unspecified behavior.

#### Fix

Verify that global objects are initialized before their first use.

#### Example

In this example, the file `aSource.cpp` contains the variable `int tacos`. The value of `int tacos` relies on the initialization of `TacoOrder t`. However, `TacoOrder t` might not be initialized by `get_default_taco()` in the source file `otherSource.cpp` before `t.get_num_tacos()` is called inside the source file `aSource.cpp`.

`aHeader.h`:

```
class TacoOrder {
    int numTacos;

public:
    TacoOrder() : numTacos(2) {}
    explicit TacoOrder(int numTacos) : numTacos(numTacos) {}

    int get_num_tacos() const { return numTacos; }
};
```

`aSource.cpp`:

```
#include "aHeader.h"
#include <iostream>

extern TacoOrder t;
int tacos = t.get_num_tacos();

void main()
{
    std::cout << tacos << std::endl;
}
```

`otherSource.cpp`:

```
#include "aHeader.h"

TacoOrder get_default_taco() { return TacoOrder(6); }
TacoOrder t = get_default_taco();
```

#### Correction

In this example, the only file that needs to be changed is `aSource.cpp`.

aHeader.h:

```
class TacoOrder {
    int numTacos;

public:
    TacoOrder() : numTacos(2) {}
    explicit TacoOrder(int numTacos) : numTacos(numTacos) {}

    int get_num_tacos() const { return numTacos; }
};
```

aSource.cpp:

```
#include "aHeader.h"
#include <iostream>

int &get_num_tacos() {
    extern TacoOrder t;
    static int tacos = t.get_num_tacos();
    return tacos;
}

void main()
{
    std::cout << get_num_tacos << std::endl;
}
```

otherSource.cpp:

```
#include "aHeader.h"

TacoOrder get_default_taco() { return TacoOrder(6); }
TacoOrder t = get_default_taco();
```

The initialization of `tacos` happens after the declaration when you move the `static int tacos` object inside the body of a function. `TacoOrder t` is initialized by `get_default_taco()` in source file `otherSource.cpp` by the time `get_num_tacos()` is called inside of `main()`.

## Check Information

**Group:** Rule 01. Declarations and Initialization (DCL)

# Version History

**Introduced in R2022b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

DCL56-CPP

# CERT C++: DCL57-CPP

Do not let exceptions escape from destructors or deallocation functions

## Description

### Rule Definition

*Do not let exceptions escape from destructors or deallocation functions.*

### Polyspace Implementation

The rule checker checks for **Class destructor exiting with an exception**.

## Examples

### Class destructor exiting with an exception

#### Issue

The checker flags:

- Explicit `throw` statements in the body of a destructor outside of a `try-catch` block. If the destructor calls another function, the checker does not detect if the called function raises an exception.
- The exception specification `noexcept(false)` in the declaration of the destructor.

The checker does not detect:

- A `catch` statement that does not catch exceptions of all types that are thrown.

  The checker considers the presence of a `catch` statement corresponding to a `try` block as an indication that an exception is caught.
- `throw` statements inside `catch` blocks.

#### Risk

Destructors are invoked at the end of code execution. When exceptions arise at this stage, they become unhandled. When such unhandled exceptions arise, depending on the hardware and software that you use, the compiler might abruptly terminate the program execution without deleting the objects in stack. Such abrupt termination might result in a resource leak and security vulnerabilities.

#### Fix

To avoid this issue:

- Declare destructors as `noexcept(true)`.
- Handle exceptions that might arise in destructors by using a `try-catch` block that includes a `catch(...)` block.

#### Example

```
#include<stdexcept>
class C {
    //...
    ~C() noexcept(false) { //Noncompliant
        //...
        throw std::logic_error("Error"); //Noncompliant
    }

};
```

In this example, the destructor of class `C` is specified as `noexcept(false)`. Polyspace flags the declaration. The destructor contains an explicit `throw` statement without encasing it in a `try-catch` block. Polyspace flags the `throw` statement.

#### Correction

One possible correction is to declare destructors as `noexcept(true)`, and then encase any `throw` statement in a `try-catch` block.

```
#include<stdexcept>
class C {
    //...
    ~C() noexcept(true) { //Compliant
        //...
        try{
            throw std::logic_error("Error"); //Compliant
        }catch(...){

        }
    }

};
```

## Check Information

**Group:** 01. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL57-CPP

# CERT C++: DCL58-CPP

Do not modify the standard namespaces

## Description

### Rule Definition

*Do not modify the standard namespaces.*

### Polyspace Implementation

The rule checker checks for **Modification of standard namespaces**.

## Examples

### Modification of standard namespaces

#### Issue

**Modification of standard namespaces** occurs when you make additions to the namespaces `std`, `posix`, or their subspaces, or you specialize class or function templates from these namespaces.

#### Risk

Adding declarations or definitions to namespace `std` or its subspaces, or to `posix` or its subspaces, leads to undefined behavior. Likewise, explicitly specializing a member function or member class of a standard library leads to undefined behavior.

The standard allows exceptions to the specialization aspect of the rule for standard library templates that require a user-defined type. If you have a process that all rule violations must be justified and an issue flagged by the checker belongs to this category of exceptions, justify the issue using comments in your result or code. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

## Check Information

**Group:** 01. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**
"Check for and Review Coding Standard Violations"

**External Websites**
DCL58-CPP

# CERT C++: DCL59-CPP

Do not define an unnamed namespace in a header file

## Description

### Rule Definition

*Do not define an unnamed namespace in a header file.*

### Polyspace Implementation

The rule checker checks for **Unnamed namespace in header file**.

## Examples

### Unnamed namespace in header file

#### Issue

**Unnamed namespace in header file** detects an unnamed namespace in a header file, which can lead to multiple definitions of objects in the namespace.

#### Risk

According to the C++ standard, names in an unnamed namespace, for instance, `aVar` here:

```
namespace {
   int aVar;
}
```

have internal linkage by default. If a header file contains an unnamed namespace, each translation unit `#include`-ing the header file defines its own instance of objects in the namespace. The multiple definitions are probably not what you intended and can lead to unexpected results, undesired memory usage or inadvertently violating the one-definition rule.

#### Fix

Specify names for namespaces in header files or avoid using namespaces in header files.

#### Example – Unexpected Results from Unnamed Namespaces in Header Files

Header File: `aHeader.h`

```
namespace { //Noncompliant
   int aVar;
}
```

First source file: `aSource.cpp`

```
#include "aHeader.h"
#include <iostream>

void setVar(int arg) {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = arg;
    std::cout << "Value set at: " << aVar << std::endl;
}
```

Second source file: `anotherSource.cpp`

```
#include "aHeader.h"
#include <iostream>

extern void setVar(int);

void resetVar() {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = 0;
    std::cout << "Value set at: 0" << std::endl;
}

void main() {
    setVar(1);
    resetVar();
}
```

In this example, the unnamed namespace leads to two definitions of `aVar` in the translation unit from `aSource.cpp` and the translation unit from `anotherSource.cpp`. The two definitions lead to the possibly unexpected output:

```
Current value: 0
Value set at: 1
Current value: 0
Value set at: 0
```

#### Correction – Avoid the Unnamed Namespace

One possible correction is to simply avoid a namespace in the header file.

Header File: `aHeader.h`

```
extern int aVar;
```

First source file: `aSource.cpp`

```
#include "aHeader.h"
#include <iostream>

void setVar(int arg) {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = arg;
    std::cout << "Value set at: " << aVar << std::endl;
}
```

Second source file: `anotherSource.cpp`

```
#include "aHeader.h"
#include <iostream>

extern void setVar(int);
int aVar;

void resetVar() {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = 0;
    std::cout << "Value set at: 0" << std::endl;
}

void main() {
    setVar(1);
    resetVar();
}
```

You now see the expected sequence in the output:

```
Current value: 0
Value set at: 1
Current value: 1
Value set at: 0
```

### Check Information

**Group:** 01. Declarations and Initialization (DCL)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

DCL59-CPP

# CERT C++: DCL60-CPP

Obey the one-definition rule

## Description

### Rule Definition

*Obey the one-definition rule.*

### Polyspace Implementation

This checker checks for:

- **Inline constraint not respected**
- **Nonidentical definitions of function or object across modules**

## Examples

### Inline Constraint Not Respected

#### Issue

**Inline constraint not respected** occurs when you refer to a file scope modifiable static variable or define a local modifiable static variable in a nonstatic inlined function. The checker considers a variable as modifiable if it is not `const`-qualified.

For instance, `var` is a modifiable `static` variable defined in an `inline` function `func`. `g_step` is a file scope modifiable static variable referred to in the same inlined function.

```
static int g_step;
inline void func (void) {
   static int var = 0;
   var += g_step;
}
```

#### Risk

When you modify a static variable in multiple function calls, you expect to modify the same variable in each call. For instance, each time you call `func`, the same instance of `var1` is incremented but a separate instance of `var2` is incremented.

```
void func(void) {
   static var1 = 0;
   var2 = 0;
   var1++;
   var2++;
}
```

If a function has an inlined and a noninlined definition in separate files, when you call the function, the C standard allows compilers to use either the inlined or the noninlined form (see ISO/IEC 9899:2011, sec. 6.7.4). If your compiler uses an inlined definition in one call and the noninlined definition in another, you are no longer modifying the same variable in both calls. This behavior defies the expectations from a static variable.

### Fix

Use one of these fixes:

- If you do not intend to modify the variable, declare it as `const`.

  If you do not modify the variable, there is no question of unexpected modification.
- Make the variable non-`static`. Remove the `static` qualifier from the declaration.

  If the variable is defined in the function, it becomes a regular local variable. If defined at file scope, it becomes an extern variable. Make sure that this change in behavior is what you intend.
- Make the function `static`. Add a `static` qualifier to the function definition.

  If you make the function `static`, the file with the inlined definition uses the inlined definition when the function is called. Other files use another definition of the function. The compiler does not decide which function definition is used.

### Example - Static Variable Use in Inlined and External Definition

```
/* file1. c  : contains inline definition of get_random()*/

inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef; //Noncompliant
    static unsigned int m_w = 0xbaddecaf; //Noncompliant

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}
```

```
/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

In this example, get_random() has an inline definition in file1.c and an external definition in file2.c. When get_random is called in file1.c, compilers can to choose whether to use the inline or the external definition.

Depending on the definition used, you might or might not modify the version of m_z and m_w in the inlined version of get_random(). This behavior contradicts the usual expectations from a static variable. When you call get_random(), you expect to modify the same m_z and m_w.

**Correction — Make Inlined Function Static**

One possible correction is to make the inlined get_random() static. Irrespective of your compiler, calls to get_random() in file1.c then use the inlined definition. Calls to get_random() in other files use the external definition. This fix removes the ambiguity about which definition is used and whether the static variables in that definition are modified.

```
/* file1. c  : contains inline definition of get_random()*/

static inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef; //Compliant
    static unsigned int m_w = 0xbaddecaf; //Compliant

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

### Nonidentical Definitions of Function or Object Across Modules

#### Issue

**Nonidentical definitions of function or object across modules** occurs when a function or object is defined in multiple modules, but with differences in tokens like identifiers, keywords, literals, operators, punctuators, and other separators. The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` or `extern` functions
- Uncalled and undefined local functions
- Unused types and variables

*The checker does not flag this issue in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

#### Risk

Having different definitions of the same object or noninlined function in different modules results in unexpected behavior. The program might crash and leak memory depending on the software and hardware that you use.

#### Fix

Define objects and noninlined functions without any differences in tokens. Use the same sequence and types of tokens in the definitions of objects and noninline functions across modules.

#### Example: Definition of Object Has Token Difference

This example uses two files:

- `file1.cpp`:

```
typedef struct S
{
   int x;
   int y;
}S;
void foo(S& s){
//...
}
```

- `file2.cpp`:

```
typedef struct S //Noncompliant
{
   int y;
   int x;
}S ;
void bar(S& s){
//...
}
```

In this example, both `file1.cpp` and `file2.cpp` define the structure `S`. The definitions switch the order of the structure fields.

#### Correction: Use Identical Definition Across Modules

One possible correction is to define the structure `S` in a header file and include the header in the two modules.

- `S.h`:

  ```
  struct S //Compliant
  {
     int x;
     int y;
  };
  ```
- `file1.cpp`:

  ```
  #include"S.h"
  void foo(S& s){
  //...
  }
  ```
- `file2.cpp`:

  ```
  #include"S.h"
  void bar(S& s){
  //...
  }
  ```

## Check Information

**Group:** 01. Declarations and Initialization (DCL)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

DCL60-CPP

# CERT C++: EXP34-C

Do not dereference null pointers

## Description

### Rule Definition

*Do not dereference null pointers.*

### Polyspace Implementation

The rule checker checks for **Null pointer**.

### Extend Checker

A default Bug Finder analysis might not flag a **Null pointer** issue when the input values are unknown and only a subset of inputs cause the issue. To check for a **Null pointer** issue caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Null pointer

**Issue**

**Null pointer** occurs when you use a pointer with a value of NULL as if it points to a valid memory location.

**Risk**

Dereferencing a null pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

**Fix**

Check a pointer for NULL before dereference.

If the issue occurs despite an earlier check for NULL, look for intermediate events between the check and the subsequent dereference. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

**Example - Null pointer error**

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 int* p=NULL;
 *p=arr[0];  //Noncompliant
 /* Defect: Null pointer dereference */

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

The pointer `p` is initialized with value of `NULL`. However, when the value `arr[0]` is written to `*p`, `p` is assumed to point to a valid memory location.

#### Correction — Assign Address to Null Pointer Before Dereference

One possible correction is to initialize `p` with a valid memory address before dereference.

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 /* Fix: Assign address to null pointer */
 int* p=&arr[0];

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

EXP34-C

# CERT C++: EXP35-C

Do not modify objects with temporary lifetime

## Description

### Rule Definition

*Do not modify objects with temporary lifetime.*

### Polyspace Implementation

The rule checker checks for **Accessing object with temporary lifetime**.

## Examples

### Accessing object with temporary lifetime

#### Issue

**Accessing object with temporary lifetime** occurs when you attempt to read from or write to an object with temporary lifetime that is returned by a function call. In a structure or union returned by a function, and containing an array, the array members are temporary objects. The lifetime of temporary objects ends:

- When the full expression or full declarator containing the call ends, as defined in the C11 Standard.
- After the next sequence point, as defined in the C90 and C99 Standards. A sequence point is a point in the execution of a program where all previous evaluations are complete and no subsequent evaluation has started yet.

For C++ code, **Accessing object with temporary lifetime** raises a defect only when you write to an object with a temporary lifetime.

If the temporary lifetime object is returned by address, no defect is raised.

#### Risk

Modifying objects with temporary lifetime is undefined behavior and can cause abnormal program termination and portability issues.

#### Fix

Assign the object returned from the function call to a local variable. The content of the temporary lifetime object is copied to the variable. You can now modify it safely.

#### Example - Modifying Temporary Lifetime Object Returned by Function Call

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6
```

```
struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

/* func_temp() returns a struct value containing
* an array with a temporary lifetime.
*/
int func(void) {

/*Writing to temporary lifetime object is
 undefined behavior
 */
    return ++(func_temp().a[0]); //Noncompliant
}

void main(void) {
    (void)func();
}
```

In this example, `func_temp()` returns by value a structure with an array member `a`. This member has temporary lifetime. Incrementing it is undefined behavior.

**Correction — Assign Returned Value to Local Variable Before Writing**

One possible correction is to assign the return of the call to `func_temp()` to a local variable. The content of the temporary object `a` is copied to the variable, which you can safely increment.

```
 #include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

int func(void) {

/* Assign object returned by function call to
 *local variable
 */
    struct S_Array s = func_temp();

/* Local variable can safely be
 *incremented
 */
    ++(s.a[0]);
    return s.a[0];
}

void main(void) {
    (void)func();
}
```

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP35-C

# CERT C++: EXP36-C

Do not cast pointers into more strictly aligned pointer types

## Description

### Rule Definition

*Do not cast pointers into more strictly aligned pointer types.*

### Polyspace Implementation

The rule checker checks for **Source buffer misaligned with destination buffer**.

## Examples

### Source buffer misaligned with destination buffer

#### Issue

The issue **Source buffer misaligned with destination buffer** occurs when the source pointer and the target pointer in a pointer-to-pointer conversion do not align. The misalignment occurs for reasons such as:

- The source pointer points to a buffer that is smaller than what the target pointer points to.
- The source pointer points to a buffer that is larger than what the target pointer points to, but the buffer size is not an exact multiple of the destination buffer size.

Polyspace does not report violations of this rule if:

- You use an environment that allows conversion between misaligned pointers. For instance, in 32bit and 64bit `x86` environment, converting `char*` into `uint16_t*` does not terminate a program abnormally.
- You use a source pointer that aligns correctly to the target type. For instance:
  - You use the `alignas` specifier to match the alignment of the source pointer with that of the target pointer.
  - You use pointers returned by functions such as `aligned_alloc()`, `malloc()`, and `realloc()`. These pointers matches the alignment of the target pointer.

#### Risk

If the alignment of a pointer changes in a pointer-to-pointer conversion, dereferencing the converted pointer causes abnormal program termination.

#### Fix

Avoid changing the alignment of a pointer in a pointer-to-pointer conversion.

#### Example — Change in Pointer Alignment During Conversion

```
#include <string.h>
struct record {
  int len;
  /* ... */
};

int copyBuffer (char *data, int offset)
{
  struct record *tmp;
  struct record dest;
  tmp = (struct record *) (data + offset);   //Noncompliant
  memcpy (&dest, tmp, sizeof (dest));
  /* ... */

  return dest.len;
}
```

In this example, the function `copyBuffer` converts a `char*` pointer into a `struct record*` pointer, followed by a `memcpy` operation. The `memcpy` operation assumes a `struct record*` alignment of `tmp`, which leads to undefined behavior. The rule checker reports a violation of this rule.

To avoid the issue, use `data + offset` as the source argument of the `memcpy` operation instead of using an intermediate `struct record*` pointer.

#### Example — Use alignas Specifier to Avoid Alignment Mismatch

This example aligns the `char` pointer `char_p` as an `int` by using the `alignas` specifier. Because `char_p` is aligned as an `int`, converting it to an `int*` does not violate this rule.

```
#include <stdalign.h>
#include <assert.h>

void foo(void) {
  /* Align char_p to the alignment of an int */
  alignas(int) char char_p = 'a';
  int *int_p = (int *)&char_p; //Compliant
  char *char_new = (char *)int_p;
  /* Both char_new and &char_p point to equally aligned objects */

}
```

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

EXP36-C

# CERT C++: EXP37-C

Call functions with the correct number and type of arguments

## Description

### Rule Definition

*Call functions with the correct number and type of arguments.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Bad file access mode or status**.
- **Unreliable cast of function pointer**.
- **Standard function call with incorrect arguments**.
- **Function declaration mismatch**
- **Incompatible Argument**

## Examples

### Bad file access mode or status

#### Issue

**Bad file access mode or status** occurs when you use functions in the `fopen` or `open` group with invalid or incompatible file access modes, file creation flags, or file status flags as arguments. For instance, for the `open` function, examples of valid:

- Access modes include `O_RDONLY`, `O_WRONLY`, and `O_RDWR`
- File creation flags include `O_CREAT`, `O_EXCL`, `O_NOCTTY`, and `O_TRUNC`.
- File status flags include `O_APPEND`, `O_ASYNC`, `O_CLOEXEC`, `O_DIRECT`, `O_DIRECTORY`, `O_LARGEFILE`, `O_NOATIME`, `O_NOFOLLOW`, `O_NONBLOCK`, `O_NDELAY`, `O_SHLOCK`, `O_EXLOCK`, `O_FSYNC`, `O_SYNC` and so on.

The defect can occur in the following situations.

| Situation | Risk | Fix |
|---|---|---|
| You pass an empty or invalid access mode to the `fopen` function.<br>According to the ANSI C standard, the valid access modes for `fopen` are:<br>• `r`,`r+`<br>• `w`,`w+`<br>• `a`,`a+`<br>• `rb`, `wb`, `ab`<br>• `r+b`, `w+b`, `a+b`<br>• `rb+`, `wb+`, `ab+` | `fopen` has undefined behavior for invalid access modes.<br>Some implementations allow extension of the access mode such as:<br>• GNU: `rb+cmxe,ccs=utf`<br>• Visual C++: `a+t`, where `t` specifies a text mode.<br>However, your access mode string must begin with one of the valid sequences. | Pass a valid access mode to `fopen`. |
| You pass the status flag `O_APPEND` to the `open` function without combining it with either `O_WRONLY` or `O_RDWR`. | `O_APPEND` indicates that you intend to add new content at the end of a file. However, without `O_WRONLY` or `O_RDWR`, you cannot write to the file.<br>The `open` function does not return -1 for this logical error. | Pass either `O_APPEND\|O_WRONLY` or `O_APPEND\|O_RDWR` as access mode. |
| You pass the status flags `O_APPEND` and `O_TRUNC` together to the `open` function. | `O_APPEND` indicates that you intend to add new content at the end of a file. However, `O_TRUNC` indicates that you intend to truncate the file to zero. Therefore, the two modes cannot operate together.<br>The `open` function does not return -1 for this logical error. | Depending on what you intend to do, pass one of the two modes. |
| You pass the status flag `O_ASYNC` to the `open` function. | On certain implementations, the mode `O_ASYNC` does not enable signal-driven I/O operations. | Use the `fcntl(pathname, F_SETFL, O_ASYNC);` instead. |

**Fix**

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Access Mode with fopen

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "rw"); //Noncompliant
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

In this example, the access mode `rw` is invalid. Because `r` indicates that you open the file for reading and `w` indicates that you create a new file for writing, the two access modes are incompatible.

#### Correction — Use Either r or w as Access Mode

One possible correction is to use the access mode corresponding to what you intend to do.

```
#include <stdio.h>

void func(void) {
    FILE *file = fopen("data.txt", "w");
    if(file!=NULL) {
        fputs("new data",file);
        fclose(file);
    }
}
```

### Unreliable cast of function pointer

#### Issue

**Unreliable cast of function pointer** occurs when a function pointer is cast to another function pointer that has a different argument or return type.

#### Risk

If you cast a function pointer to another function pointer that has a different argument or return type, and then use the latter function pointer to call a function, the behavior is undefined.

#### Fix

Avoid a cast between two function pointers that have a mismatch in argument or return types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.

- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Unreliable Cast of Function Pointer Error

```
int f(char c) {
    return c;
}

int g(int i) {
    return i;
}

typedef int (*fptr_t)(char);
typedef int (*gptr_t)(int);

void call() {
    gptr_t ptr = (gptr_t) f;//Noncompliant
    int i = ptr(511); // Undefined behavior
}
```

In this example, the pointer to function `f` is cast to `gptr_t`, which is the type of the function `g`. When the function pointer is used to call `f` by using integers, the code behavior is undefined. Polyspace flags the unreliable cast of the function pointer.

#### Correction — Avoid Function Pointer Cast

To avoid undefined behavior, refactor your code so that the function `f` is not cast into a different argument type. For instance:

```
 int f(int c) { //Fix: declare f with int argument
    return c;
}

int g(int i) {
    return i;
}

typedef int (*fptr_t)(char);
typedef int (*gptr_t)(int);

void call() {
    gptr_t ptr = (gptr_t) f;//Compliant
    int i = ptr(511);
}
```

### Standard function call with incorrect arguments

#### Issue

**Standard function call with incorrect arguments** occurs when the arguments to certain standard functions do not meet the requirements for their use in the functions.

For instance, the arguments to these functions can be invalid in the following ways.

| Function Type | Situation | Risk | Fix |
|---|---|---|---|
| String manipulation functions such as `strlen` and `strcpy` | The pointer arguments do not point to a `NULL`-terminated string. | The behavior of the function is undefined. | Pass a `NULL`-terminated string to string manipulation functions. |
| File handling functions in `stdio.h` such as `fputc` and `fread` | The `FILE*` pointer argument can have the value `NULL`. | The behavior of the function is undefined. | Test the `FILE*` pointer for `NULL` before using it as function argument. |
| File handling functions in `unistd.h` such as `lseek` and `read` | The file descriptor argument can be -1. | The behavior of the function is undefined.<br>Most implementations of the `open` function return a file descriptor value of -1. In addition, they set `errno` to indicate that an error has occurred when opening a file. | Test the return value of the `open` function for -1 before using it as argument for `read` or `lseek`.<br>If the return value is -1, check the value of `errno` to see which error has occurred. |
| | The file descriptor argument represents a closed file descriptor. | The behavior of the function is undefined. | Close the file descriptor only after you have completely finished using it. Alternatively, reopen the file descriptor before using it as function argument. |
| Directory name generation functions such as `mkdtemp` and `mkstemps` | The last six characters of the string template are not `XXXXXX`. | The function replaces the last six characters with a string that makes the file name unique. If the last six characters are not `XXXXXX`, the function cannot generate a unique enough directory name. | Test if the last six characters of a string are `XXXXXX` before using the string as function argument. |
| Functions related to environment variables such as `getenv` and `setenv` | The string argument is `""`. | The behavior is implementation-defined. | Test the string argument for `""` before using it as `getenv` or `setenv` argument. |
| | The string argument terminates with an equal sign, `=`. For instance, `"C="` instead of `"C"`. | The behavior is implementation-defined. | Do not terminate the string argument with `=`. |

| Function Type | Situation | Risk | Fix |
|---|---|---|---|
| String handling functions such as `strtok` and `strstr` | • `strtok`: The delimiter argument is `""`.<br>• `strstr`: The search string argument is `""`. | Some implementations do not handle these edge cases. | Test the string for `""` before using it as function argument. |

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - `NULL` Pointer Passed as `strnlen` Argument

```
#include <string.h>
#include <stdlib.h>

enum {
    SIZE10 = 10,
    SIZE20 = 20
};

int func() {
    char* s = NULL;
    return strnlen(s, SIZE20); //Noncompliant
}
```

In this example, a `NULL` pointer is passed as `strnlen` argument instead of a `NULL`-terminated string.

Before running analysis on the code, specify a GNU compiler. See `Compiler (-compiler)`.

#### Correction — Pass `NULL`-terminated String

Pass a `NULL`-terminated string as the first argument of `strnlen`.

```
#include <string.h>
#include <stdlib.h>

enum {
    SIZE10 = 10,
    SIZE20 = 20
};
```

```
int func() {
    char* s = "";
    return strnlen(s, SIZE20);
}
```

### Function declaration mismatch

#### Issue

**Note** In C++ code, this checker applies to functions that are specified as `extern "C"`.

**Function declaration mismatch** occurs when the prototype of a `extern "C"` function does not match its definition. Type mismatch between the arguments of the function definition and the function prototype might depend on your environment. Polyspace considers two types as compatible if they have the same size and signedness in the environment that you use. For instance, if your specify `-target` as i386, Polyspace considers `long` and `int` as compatible types.

In C++, if a function is not specified as `extern "C"` and its prototype does not match any function definition, the compiler treats the prototype as that of an undefined overload of the function. Polyspace does not flag calls to such undefined functions.

*The checker does not flag this issue in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

#### Risk

Function declaration mismatch might result in undefined behavior. When function declarations are specified with `extern "C"`, mismatches between definition and declaration of a function might produce only warnings during compilation, resulting in code that compiles but behaves in an unexpected way.

#### Fix

- Before you call a function, provide its complete prototype, even if you define the function later in the same file.
- Avoid any mismatch between the number arguments in the function prototype declaration and the function definition.
- Avoid any mismatch between the argument types of the function prototype declaration and the function definition.

#### Example — Noncompliant Function Calls

```
// file1.c
extern "C" void foo(int iVar){
    //...
}
extern "C" void bar(int iVar){
    //...
}
extern "C" void fubar(int A, ...){
    //...
}
```

```
//prototype.h
extern "C" void foo(void);
extern "C" void fubar(int A, ...);
extern "C" void bar(long iVar);
```

```
//file2.c
//file2.c
#include"prototype.h"
void call_funcs(){
    int iTemp;
    float fTemp;
    long lTemp;
    foo(); //Noncompliant
    bar(lTemp);//Noncompliant in x86_64
    fubar(iTemp,fTemp);//Compliant

}
```

In this example, the functions `foo`, `bar`, and `fubar` are defined in the file `file1.c`. Their prototypes are declared in `prototype.h`. These functions are then called in the file `file2.c`.

- The function `foo` is defined with an `int` argument but its prototype is declared without any argument. Because of this mismatch, Polyspace flags the function call.
- The function `bar` is defined with an `int` argument but its prototype is declared with a `long` argument. These two types are not compatible in `x86_64` environment. When you specify `-target` as `x86_64`, Polyspace flags the function call.
- The call to the variadic function `fubar` is compliant because its call signature, prototype, and definition matches.

#### Correction — Compliant Function Calls

The fix for this defect is to declare complete and accurate prototypes for the called functions. In this case, fix the raised issues by resolving the mismatches between the function definition and prototype declaration. Update the function calls to match the updated prototypes.

```
// file1.c
extern "C" void foo(int iVar){
    //...
}
extern "C" void bar(int iVar){
    //...
}
extern "C" void fubar(int A, ...){
    //...
}
```

```
//prototype.h
extern "C" void foo(int);
extern "C" void fubar(int A, ...);
extern "C" void bar(int iVar);
```

```
//file2.c
//file2.c
#include"prototype.h"
void call_funcs(){
    int iTemp;
    float fTemp;
    long lTemp;
    foo(iTemp); //Compliant
    bar(iTemp);//Compliant in x86_64
    fubar(iTemp,fTemp);//Compliant

}
```

### Incompatible Argument

#### Issue

**Incompatible Argument** occurs when an external function is called by using an argument that is not compatible with the prototype. The compatibility of types might depend on the set of hardware and software that you use. For instance, consider this code:

```
extern long foo(int);
```

```
long bar(long i) {
    return foo(i); //Noncompliant: calls foo(int) with a long
}
```

The external function `foo` is called with a `long` when an `int` is expected. In environments where the size of an `int` is smaller than the size of a `long`, this function call is incompatible with the prototype, resulting in a defect.

In C++, this defect might cause a compilation error.

#### Risk

Calling external functions with arguments that are incompatible with the parameter is undefined behavior. Depending on your environment, the code might compile but behave in an unexpected way.

#### Fix

When calling external functions, use argument types that are smaller or equal in size compared to the parameter type defined in the prototype. Check the sizes of various integer types in your environment to determine compatibility of argument and parameter types.

#### Example — Call External Functions with Incompatible argument

```
extern long foo1(int);
extern long foo2(long);
void bar(){
    int varI;
    long varL;
    foo1(varL);//Noncompliant
    foo2(varI);//Compliant
}
```

In this example, the external function `foo1` is called with a `long` argument, while the prototype specifies the parameter as an `int`. In `x86` architecture, the size of `long` is larger than the size of `int`. The call `foo1(varL)` might result in undefined behavior. Polyspace flags the call. The call `foo2(varI)` uses an `int` argument while the parameter is specified as a `long`. This type of mismatch is compliant with this rule because the size of `int` is not larger than the size of `long`.

To run this example in Polyspace, use these options:

- `-target x86_64`

See `Target processor type (-target)`.

#### Correction — Cast Variables Explicitly to Match Argument to Parameter

To fix this issue, cast the argument of `foo1` explicitly so that argument type and parameter type matches.

```
extern long foo1(int);
extern long foo2(long);
void bar(){
    int varI;
    long varL;
    foo1((int)varL);//Compliant
    foo2(varI);//Compliant
}
```

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP37-C

# CERT C++: EXP39-C

Do not access a variable through a pointer of an incompatible type

## Description

### Rule Definition

*Do not access a variable through a pointer of an incompatible type.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Pointer conversion to unrelated pointer type**
- **Reading memory reallocated from object of another type without reinitializing first**

.

## Examples

### Pointer conversion to unrelated pointer type

#### Issue

This issue occurs when you convert a pointer to an unrelated pointer type. The checker flags all pointer conversions including between a pointer to a `struct` object and a pointer to the first member of the same `struct` type. Indirect conversions from a pointer to non-pointer type are not detected.

#### Risk

The outcome of the conversion between pointers of unrelated types is unspecified in the C standard. Such conversion might result in unexpected behavior.

#### Fix

Avoid converting pointers to unrelated types. Refactor your code and logic to reduce the necessity of pointer conversions.

### Reading memory reallocated from object of another type without reinitializing first

#### Issue

This issue occurs when you do the following in sequence:

1. Reallocate memory to an object with a type that is different from the original allocation.

   For instance, in this code snippet, a memory originally allocated to a pointer with type `struct A*` is reallocated to a pointer with type `struct B*`:

   ```
   struct A;
   struct B;

   struct A *Aptr = (struct A*) malloc(sizeof(struct A));
   struct B *Bptr = (struct B*) realloc(Aptr, sizeof(struct B));
   ```

2 Read from this reallocated memory without reinitializing the memory first.

Read accesses on the pointer to the reallocated memory can happen through pointer dereference or array indexing. Passing the pointer to a function that takes a pointer to a `const`-qualified object as the corresponding parameter also counts as a read access.

**Risk**

Reading from reallocated memory that has not been reinitialized is undefined behavior.

**Fix**

Reinitialize memory after reallocation and before the first read access.

The checker considers any write access on the pointer to the reallocated memory as satisfying the reinitialization requirement (even if the object might only be partially reinitialized). Write accesses on the pointer to the reallocated memory can happen through pointer dereference or array indexing. Passing the pointer to a function that takes a pointer to a non-`const`-qualified object as the corresponding parameter also counts as a write access.

**Example – Noncompliant: Reading from Reallocated Memory Without Reinitializing First**

```
#include<cstdlib>

struct group {
    char *groupFirst;
    int groupSize;
};

struct groupWithID {
    int groupID;
    char *groupFirst;
    int groupSize;
};

char* readName();
int readSize();

void createGroup(int nextAvailableID) {
    struct group *aGroup;
    struct groupWithID *aGroupWithID;

    aGroup = (struct group*) malloc(sizeof(struct group));
    aGroup->groupFirst = readName();
    aGroup->groupSize  = readSize();

    if(!aGroup) {
        /*Handle error*/
    }

    /* Reassign to group with ID */
    aGroupWithID = (struct groupWithID*) realloc(aGroup, sizeof(struct groupWithID));
    if(!aGroupWithID) {
        free(aGroup);
        /*Handle error*/
    }

    if(aGroupWithID -> groupSize > 0) { /* Noncompliant */
```

```
        /* ... */
    }

    /* ...*/
    free(aGroupWithID);
}
```

In this example, the memory allocated to a `group*` pointer using the `malloc` function is reallocated to a `groupWithID*` pointer using the `realloc` function. There is a read access on the reallocated memory before the memory is reinitialized.

**Correction – Reinitialize Memory After Reallocation and Before First Read**

Reinitialize the memory assigned to the `groupWithID*` pointer before the first read access. All bits of the memory can be reinitialized using the `memset` function.

```
#include<cstdlib>
#include<cstring>

struct group {
    char *groupFirst;
    int groupSize;
};

struct groupWithID {
    int groupID;
    char *groupFirst;
    int groupSize;
};

char* readName();
int readSize();

void createGroup(int nextAvailableID) {
    struct group *aGroup;
    struct groupWithID *aGroupWithID;

    aGroup = (struct group*) malloc(sizeof(struct group));
    aGroup->groupFirst = readName();
    aGroup->groupSize  = readSize();

    if(!aGroup) {
        /*Handle error*/
    }

    /* Reassign to group with ID */
    aGroupWithID = (struct groupWithID*) realloc(aGroup, sizeof(struct groupWithID));
    if(!aGroupWithID) {
        free(aGroup);
        /*Handle error*/
    }

    memset(aGroupWithID, 0 , sizeof(struct groupWithID));
    /* Reinitialize group */
    if(aGroupWithID -> groupSize > 0) {
        /* ... */
    }
    /* ...*/
    free(aGroupWithID);
}
```

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

EXP39-C

# CERT C++: EXP42-C

Do not compare padding data

## Description

### Rule Definition

*Do not compare padding data.*

### Polyspace Implementation

The rule checker checks for **Memory comparison of padding data**.

## Examples

### Memory comparison of padding data

#### Issue

**Memory comparison of padding data** occurs when you use the `memcmp` function to compare two structures as a whole. In the process, you compare meaningless data stored in the structure padding.

For instance:

```
struct structType {
    char member1;
    int member2;
    //...
    //...
};

structType var1;
structType var2;
//...
//...
if(memcmp(&var1,&var2,sizeof(var1)))
{...}
```

#### Risk

If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. For an example of padding, see `Higher Estimate of Size of Local Variables`.

The content of these extra padding bytes is meaningless. The C Standard allows the content of these bytes to be indeterminate, giving different compilers latitude to implement their own padding. If you perform a byte-by-byte comparison of structures with `memcmp`, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.

#### Fix

Instead of comparing two structures in one attempt, compare the structures member by member.

For efficient code, write a function that does the comparison member by member. Use this function for comparing two structures.

You can use `memcmp` for byte-by-byte comparison of structures only if you know that the structures do not contain padding. Typically, to prevent padding, you use specific attributes or pragmas such as `#pragma pack`. However, these attributes or pragmas are not supported by all compilers and make your code implementation-dependent. If your structures contain bit-fields, using these attributes or pragmas cannot prevent padding.

#### Example - Structures Compared with `memcmp`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{

    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    if (0 == memcmp(left, right, sizeof(S_Padding))) //Noncompliant
    {
        return 1;
    }
    else
        return 0;
}
```

In this example, `memcmp` compares byte-by-byte the two structures that `left` and `right` point to. Even if the values stored in the structure members are the same, the comparison can show an inequality if the meaningless values in the padding bytes are not the same.

#### Correction — Compare Structures Member by Member

One possible correction is to compare individual structure members.

**Note** You can compare entire arrays by using `memcmp`. All members of an array have the same data type. Padding bytes are not required to store arrays.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{
    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    return ((left->c == right->c) &&
            (left->i == right->i) &&
            (left->bf1 == right->bf1) &&
            (left->bf2 == right->bf2) &&
            (memcmp(left->buffer, right->buffer, 20) == 0));
}
```

### Check Information

**Group:** 02. Expressions (EXP)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**

“Check for and Review Coding Standard Violations”

**External Websites**

EXP42-C

# CERT C++: EXP45-C

Do not perform assignments in selection statements

## Description

### Rule Definition

*Do not perform assignments in selection statements.*

### Polyspace Implementation

The rule checker checks for **Invalid use of = (assignment) operator**.

## Examples

### Invalid use of = (assignment) operator

#### Issue

**Invalid use of = operator** occurs when an assignment is made inside the predicate of a conditional, such as `if` or `while`.

In C and C++, a single equal sign is an assignment not a comparison. Using a single equal sign in a conditional statement can indicate a typo or a mistake.

#### Risk

- Conditional statement tests the wrong values— The single equal sign operation assigns the value of the right operand to the left operand. Then, because this assignment is inside the predicate of a conditional, the program checks whether the new value of the left operand is nonzero or not NULL.
- Maintenance and readability issues — Even if the assignment is intended, someone reading or updating the code can misinterpret the assignment as an equality comparison instead of an assignment.

#### Fix

- If the assignment is a bug, to check for equality, add a second equal sign (==).
- If the assignment inside the conditional statement was intentional, to improve readability, separate the assignment and the test. Move the assignment outside the control statement. In the control statement, simply test the result of the assignment.

  If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

  - “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
  - “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
  - “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Single Equal Sign Inside an `if` Condition

```
#include <stdio.h>

void bad_equals_ex(int alpha, int beta)
{
    if(alpha = beta) //Noncompliant
    {
        printf("Equal\n");
    }
}
```

The equal sign is flagged as a defect because the assignment operator is used within the predicate of the if-statement. The predicate assigns the value `beta` to `alpha`, then implicitly tests whether `alpha` is true or false.

#### Correction — Change Expression to Comparison

One possible correction is adding an additional equal sign. This correction changes the assignment to a comparison. The if condition compares whether `alpha` and `beta` are equal.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha == beta)
    {
        printf("Equal\n");
    }
}
```

#### Correction — Assignment and Comparison Inside the `if` Condition

If an assignment must be made inside the predicate, a possible correction is adding an explicit comparison. This correction assigns the value of `beta` to `alpha`, then explicitly checks whether `alpha` is nonzero. The code is clearer.

```
#include <stdio.h>

int assignment_not_zero(int alpha, int beta)
{
    if((alpha = beta) != 0)
    {
        return alpha;
    }
    else
    {
        return 0;
    }
}
```

#### Correction — Move Assignment Outside the `if` Statement

If the assignment can be made outside the control statement, one possible correction is to separate the assignment and comparison. This correction assigns the value of `beta` to `alpha` before the if. Inside the if-condition, only `alpha` is given to test if `alpha` is nonzero or not NULL.

```
#include <stdio.h>
```

```
void assign_and_print(int alpha, int beta)
{
    alpha = beta;
    if(alpha)
    {
        printf("%d", alpha);
    }
}
```

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

EXP45-C

# CERT C++: EXP46-C

Do not use a bitwise operator with a Boolean-like operand

## Description

### Rule Definition

*Do not use a bitwise operator with a Boolean-like operand.*

### Polyspace Implementation

The rule checker checks for **Use of bitwise operator with a Boolean-like operand**.

## Examples

### Use of bitwise operator with a Boolean-like operand

#### Issue

**Use of bitwise operator with a Boolean-like operand** occurs when you use bitwise operators, such as:

- Bitwise AND (&, &=)
- Bitwise OR (|, |=)
- Bitwise XOR (^, ^=)
- Bitwise NOT(~)

with:

- Boolean type variables
- Outputs of relational or equality expressions

Using Boolean type variables as array indices, in Boolean arithmetic expression, and in shifting operations does not raise this defect.

#### Risk

Boolean-like operands, such as variables of type `bool` and outputs of relational operators typically appear in logical expressions. Using a bitwise operator in an expression containing Boolean variables and relational operators might be a sign of logic error. Because bitwise operators and logical operators look similar, you might inadvertently use a bitwise operator instead of a logical operator. Such logic errors do not raise any compilation error and can introduce bugs in your code that are difficult to find.

#### Fix

Use logical operators in expressions that contain Boolean variables and relational operator. To indicate that you intend to use a bitwise operator in such an expression, use parentheses.

#### Example — Possible Bug Due to Using Bitwise Operator

```
class User{
    //...
    int uid;
    int euid;
public:
    int getuid();
    int geteuid();
};
void Noncompliant ()
{
    User nU;
    if (nU.getuid () & nU.geteuid () == 0) {   //Noncompliant
        //...
    }else{
        //...
    }
}
```

In this example, the `if-else` block is executed conditionally. The conditional statement uses the bitwise AND (&) instead of the logical AND (&&), perhaps by mistake. Consider when the function `nU.geteuid()` evaluates to 0, and `nU.getuid()` evaluates to 2. In this case, the `else` block of code executes if you use & because `2&1` evaluates to `false`. Conversely, the `if` block of code executes when you use && because `2&&1` evaluates to `true`. Using & instead of && might introduce logic errors and bugs in your code that are difficult to find. Polyspace flags the use of bitwise operators in these kinds of expressions where relational operators are also used.

#### Correction — Use Logical Operators with Boolean-Like Operands

One possible correction is to use logical operators in expressions that contain relational operators and Boolean variables.

```
class User{
    //...
    int uid;
    int euid;
public:
    int getuid();
    int geteuid();
};
void Noncompliant ()
{
    User nU;
    if (nU.getuid () && nU.geteuid () == 0) {   //Compliant
        //...
    }else{
        //...
    }
}
```

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

EXP46-C

# CERT C++: EXP47-C

Do not call va_arg with an argument of the incorrect type

## Description

### Rule Definition

*Do not call va_arg with an argument of the incorrect type.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Incorrect data type passed to va_arg**.
- **Too many va_arg calls for current argument list**.

## Examples

### Incorrect data type passed to va_arg

#### Issue

**Incorrect data type passed to va_arg** when the data type in a va_arg call does not match the data type of the variadic function argument that va_arg reads.

For instance, you pass an unsigned char argument to a variadic function func. Because of default argument promotion, the argument is promoted to int. When you use a va_arg call that reads an unsigned char argument, a type mismatch occurs.

```
void func (int n, ...) {
   ...
   va_list args;
   va_arg(args, unsigned char);
   //...
}

void main(void) {
   unsigned char c;
   func(1,c);
}
```

#### Risk

In a variadic function (function with variable number of arguments), you use va_arg to read each argument from the variable argument list (va_list). The va_arg use does not guarantee that there actually exists an argument to read or that the argument data type matches the data type in the va_arg call. You have to make sure that both conditions are true.

Reading an incorrect type with a va_arg call can result in undefined behavior. Because function arguments reside on the stack, you might access an unwanted area of the stack.

#### Fix

Make sure that the data type of the argument passed to the variadic function matches the data type in the `va_arg` call.

Arguments of a variadic function undergo default argument promotions. The argument data types of a variadic function cannot be determined from a prototype. The arguments of such functions undergo default argument promotions (see Sec. 6.5.2.2 and 7.15.1.1 in the C99 Standard). Integer arguments undergo integer promotion and arguments of type `float` are promoted to `double`. For integer arguments, if a data type can be represented by an `int`, for instance, `char` or `short`, it is promoted to an `int`. Otherwise, it is promoted to an `unsigned int`. All other arguments do not undergo promotion.

To avoid undefined and implementation-defined behavior, minimize the use of variadic functions. Use the checkers for `MISRA C:2012 Rule 17.1` or `MISRA C++:2008 Rule 8-4-1` to detect use of variadic functions.

#### Example - `char` Used as Function Argument Type and va_arg argument

```
#include <stdarg.h>
#include <stdio.h>

unsigned char func(size_t count, ...) {
    va_list ap;
    unsigned char result = 0;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, unsigned char); //Noncompliant
    }
    va_end(ap);
    return result;
}

void func_caller(void) {
    unsigned char c = 0x12;
    (void)func(1, c);
}
```

In this example, `func` takes an `unsigned char` argument, which undergoes default argument promotion to `int`. The data type in the `va_arg` call is still `unsigned char`, which does not match the `int` argument type.

#### Correction — Use `int` as va_arg Argument

One possible correction is to read an `int` argument with `va_arg`.

```
#include <stdarg.h>
#include <stdio.h>

unsigned char func(size_t count, ...) {
    va_list ap;
    unsigned char result = 0;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
    }
    va_end(ap);
    return result;
}

void func_caller(void) {
    unsigned char c = 0x12;
    (void)func(1, c);
}
```

### Too many va_arg calls for current argument list

#### Issue

**Too many va_arg calls for current argument list** occurs when the number of calls to va_arg exceeds the number of arguments passed to the corresponding variadic function. The analysis raises a defect only when the variadic function is called.

**Too many va_arg calls for current argument list** does not raise a defect when:

- The number of calls to va_arg inside the variadic function is indeterminate. For example, if the calls are from an external source.
- The va_list used in va_arg is invalid.

#### Risk

When you call va_arg and there is no next argument available in va_list, the behavior is undefined. The call to va_arg might corrupt data or return an unexpected result.

#### Fix

Ensure that you pass the correct number of arguments to the variadic function.

#### Example - No Argument Available When Calling va_arg

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/
int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {
/* No further argument available
* in va_list when calling va_arg
*/

            result += va_arg(ap, int); //Noncompliant
        }
    }
    va_end(ap);
    return result;
}
```

```
void func(void) {

    (void)variadic_func(2, 100);

}
```

In this example, the named argument and only one variadic argument are passed to `variadic_func()` when it is called inside `func()`. On the second call to `va_arg`, no further variadic argument is available in `ap` and the behavior is undefined.

**Correction — Pass Correct Number of Arguments to Variadic Function**

One possible correction is to ensure that you pass the correct number of arguments to the variadic function.

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/

int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {

/* The correct number of arguments is
* passed to va_list when variadic_func()
* is called inside func()
*/
            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100, 200);

}
```

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

EXP47-C

# CERT C++: EXP50-CPP

Do not depend on the order of evaluation for side effects

## Description

### Rule Definition

*Do not depend on the order of evaluation for side effects.*

### Polyspace Implementation

The rule checker checks for **Expression value depends on order of evaluation**.

## Examples

### Expression value depends on order of evaluation

#### Issue

The issue occurs when the value of an expression is not the same depending on the order of evaluation of the expression.

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.

Therefore, the rule checker forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation. The rule checker also detects cases where a volatile variable is read more than once in an expression.

#### Risk

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP50-CPP

# CERT C++: EXP51-CPP

Do not delete an array through a pointer of the incorrect type

## Description

### Rule Definition

*Do not delete an array through a pointer of the incorrect type.*[9]

### Polyspace Implementation

The rule checker checks for the issue **Delete operator used to destroy downcast object of different type**.

## Examples

### Delete Operator Used to Destroy Downcast Object of Different Type

#### Issue

This issue occurs when an array object is deleted through a pointer type that is different from the pointer type of the object.

#### Risk

Deleting an array through the incorrect pointer type results in undefined behavior.

#### Fix

Match the static pointer type to the dynamic type of the object.

#### Example — Delete Downcast Object Through Different Static Pointer Type than Dynamic Object Type

```
struct Shape {
    virtual ~Shape() = default;
};

struct Rectangle : Shape {};

void example()
```

[9] *This software has been created by MathWorks incorporating portions of: the "SEI CERT-C Website," © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, "SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition," © 2016 Carnegie Mellon University, and "SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition" © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

```
{
    Shape* r = new Rectangle[10];
    // do something
    delete[] r;        //Noncompliant
}
```

In this example, the pointer to the new `Rectangle` object is stored as the pointer type `Shape*`. As `Shape` and `Rectangle` are different types, when you attempt to delete the `Rectangle` array through `Shape*`, the result is undefined behavior.

**Correction — Delete Downcast Object Through Same Static Pointer Type as Dynamic Object Type**

```
struct Shape {
    virtual ~Shape() = default;
};

struct Rectangle : Shape {};

void example()
{
    Rectangle* r = new Rectangle[10];
    // do something
    delete[] r;
}
```

By changing the pointer type from `Shape*` to `Rectangle*`, you can avoid undefined behavior when deleting the array through the pointer. This is because the static type and dynamic type of the object are the same.

## Check Information

**Group:** Rule 02. Expressions (EXP)

# Version History

**Introduced in R2022b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP51-CPP

# CERT C++: EXP52-CPP

Do not rely on side effects in unevaluated operands

## Description

### Rule Definition

*Do not rely on side effects in unevaluated operands.*

### Polyspace Implementation

The rule checker checks for the following:

- **Logical operator operand with side effects**
- **`sizeof`, `alignof` or `decltype` operand with side effects**

## Examples

### Logical operator operand with side effects

#### Issue

The issue occurs when the right hand operand of a logical && or || operator contains side effects. When evaluated, an expression with side effect modifies at least one of the variables in the expression.

The checker does not consider volatile accesses and function calls as potential side effects.

#### Risk

When evaluated, an expression with side effect modifies at least one of the variables in the expression. For instance, `n++` is an expression with side effect.

The right-hand operand of a:

- Logical && operator is evaluated only if the left-hand operand evaluates to true.
- Logical `||` operator is evaluated only if the left-hand operand evaluates to false.

In other cases, the right-hand operands are not evaluated, so side effects of the expression do not take place. If your program relies on the side effects, you might see unexpected results in those cases.

#### Fix

If you want the expression in the right-hand operand evaluated, perform the evaluation in a separate statement.

For instance, instead of:

```
if(isOK && n++) {}
```

perform the operation in two steps:

```
n++;
if(isOK && n) {}
```

### `sizeof`, `alignof`, or `decltype` operand with side effects

#### Issue

This issue occurs when the `sizeof`, `alignof` or `decltype` operator operates on an expression with a side effect. When evaluated, an expression with side effect modifies at least one of the variables in the expression.

For instance, the defect checker does not flag `sizeof(n+1)` because `n+1` does not modify `n`. The checker flags `sizeof(n++)` because `n++` is intended to modify `n`.

#### Risk

Side effects in an `alignof` operator or `decltype` operator do not persist beyond the operation. The expression in a `sizeof` operator is evaluated only if it is required for calculating the size of a variable-length array, for instance, `sizeof(a[n++])`.

When an expression with a side effect is not evaluated, the variable modification from the side effect does not happen. If you rely on the modification, you can see unexpected results.

#### Fix

Evaluate the expression with a side effect in a separate statement, and then use the result in a `sizeof`, `_Alignof`, or `_Generic` operator.

For instance, instead of:

```
a = sizeof(n++);
```

perform the operation in two steps:

```
n++;
a = sizeof(n);
```

The checker considers a function call as an expression with a side effect. Even if the function does not have side effects now, it might have side effects on later additions. The code is more maintainable if you call the function outside the `sizeof` operator. If you call a function in a `decltype`, for instance, to select the correct overload of a function and then determine its return type, the checker considers such a call as an exception.

#### Example – Increment Operator in `sizeof`

```
#include <stdio.h>

void func(void) {
    unsigned int a = 1U;
    unsigned int b = (unsigned int)sizeof(++a); //Noncompliant
    printf ("%u, %u\n", a, b);
}
```

In this example, `sizeof` operates on `++a`, which is intended to modify `a`. Because the expression is not evaluated, the modification does not happen. The `printf` statement shows that `a` still has the value `1`.

#### Correction — Perform Increment Outside `sizeof`

One possible correction is to perform the increment first, and then provide the result to the `sizeof` operator.

```
#include <stdio.h>

void func(void) {
    unsigned int a = 1U;
    ++a;
    unsigned int b = (unsigned int)sizeof (a);
    printf ("%u, %u\n", a, b);
}
```

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP52-CPP

# CERT C++: EXP53-CPP

Do not read uninitialized memory

## Description

### Rule Definition

*Do not read uninitialized memory.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Non-initialized pointer**.
- **Non-initialized variable**.

### Extend Checker

You can extend the checker in the following ways:

- Polyspace does not flag passing pointers to noninitialized variables to functions. To detect noninitialized variables that are passed to functions by pointers, extend the checker by using the option `-code-behavior-specification`. See “Extend Checkers for Initialization to Check Function Arguments Passed by Pointers”.
- If a variable in your code is non-initialized only for certain system input values, you can see one possible combination of input values causing the defect. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Non-initialized pointer

#### Issue

**Non-initialized pointer** occurs when a pointer is not assigned an address before dereference.

#### Risk

Unless a pointer is explicitly assigned an address, it points to an unpredictable location.

#### Fix

The fix depends on the root cause of the defect. For instance, you assigned an address to the pointer but the assignment is unreachable.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below. It is a good practice to initialize a pointer to NULL when declaring the pointer.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example - Non-initialized pointer error**

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
      {
        pi = new int(0);
        if (pi == NULL)
            return NULL;
      }

    *pi = j; //Noncompliant

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not. Because `pi` might be accessed without being initialized first, Polyspace flags it.

**Correction — Initialize Pointer on Every Execution Path**

One possible correction is to assign an address to `pi` in each branch of the execution path so that `pi` is not accessed before it is initialized.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
       {
        pi = new int(0);
        if (pi == NULL)
            return NULL;
       }
    /* Fix: Initialize pi in branches of if statement  */
    else
        pi = prev;


    *pi = j;//Compliant
    return pi;
}
```

### Non-initialized variable

#### Issue

**Non-initialized variable** occurs when a variable is not initialized before its value is read.

#### Risk

Unless a variable is explicitly initialized, the variable value is unpredictable. You cannot rely on the variable having a specific value.

#### Fix

The fix depends on the root cause of the defect. For instance, you assigned a value to the variable but the assignment is unreachable or you assigned a value to the variable in one of two branches of a conditional statement. Fix the unreachable code or missing assignment.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below. It is a good practice to initialize a variable at declaration.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val; //Noncompliant

}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

#### Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that the initialization is not bypassed on some execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP53-CPP

# CERT C++: EXP54-CPP

Do not access an object outside of its lifetime

## Description

### Rule Definition

*Do not access an object outside of its lifetime.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Non-initialized pointer**.
- **Non-initialized variable**.
- **Use of previously freed pointer**.
- **Pointer or reference to stack variable leaving scope**.
- **Accessing object with temporary lifetime**.

### Extend Checker

You can extend the checker in the following ways:

- Polyspace does not flag passing pointers to noninitialized variables to functions. To detect noninitialized variables that are passed to functions by pointers, extend the checker by using the option `-code-behavior-specification`. See “Extend Checkers for Initialization to Check Function Arguments Passed by Pointers”.
- If a variable in your code is non-initialized only for certain system input values, you can see one possible combination of input values causing the defect. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Non-initialized pointer

#### Issue

**Non-initialized pointer** occurs when a pointer is not assigned an address before dereference.

#### Risk

Unless a pointer is explicitly assigned an address, it points to an unpredictable location.

#### Fix

The fix depends on the root cause of the defect. For instance, you assigned an address to the pointer but the assignment is unreachable.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface".

See examples of fixes below. It is a good practice to initialize a pointer to NULL when declaring the pointer.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Non-initialized pointer error

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == nullptr)
      {
        pi = new int;
        if (pi == nullptr) return NULL;
      }
    *pi = j;  //Noncompliant

    return pi;
}
```

If `prev` is not `nullptr`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `nullptr` or not.

#### Correction — Initialize Pointer on Every Execution Path

One possible correction is to assign an address to `pi` when `prev` is not `nullptr`. Alternatively, initialize `pi` as a `nullptr` during its declaration.

```
#include <cstdlib>

int* assign_pointer(int* prev)
{
    int j = 42;
/*Fix: Initialize pointers by using nullptr during declaration*/
    int* pi = nullptr;

    if (prev == NULL)
       {
        pi = new int;
        if (pi == nullptr) return NULL;
       }
    /* Fix: Initialize pi in branches of if statement  */
    else
        pi = prev;
    *pi = j;
    return pi;
}
```

### Non-initialized variable

#### Issue

**Non-initialized variable** occurs when a variable is not initialized before its value is read.

#### Risk

Unless a variable is explicitly initialized, the variable value is unpredictable. You cannot rely on the variable having a specific value.

#### Fix

The fix depends on the root cause of the defect. For instance, you assigned a value to the variable but the assignment is unreachable or you assigned a value to the variable in one of two branches of a conditional statement. Fix the unreachable code or missing assignment.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below. It is a good practice to initialize a variable at declaration.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val; //Noncompliant
}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

#### Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that the initialization is not bypassed on some execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

### Use of previously freed pointer

#### Issue

**Use of previously freed pointer** occurs when you access a block of memory after deallocating the block, for instance, by using the `free` function or the `delete` operator.

#### Risk

When a pointer is allocated dynamic memory by using the functions `malloc`, `calloc`, `realloc` or the operator `new`, it points to a memory location on the heap. When you use the `free` function or the `delete` operator on this pointer, the associated block of memory is deallocated. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. See if you intended to deallocate the memory later or allocate another memory block to the pointer before access.

As a good practice, after you deallocate a memory block, assign the corresponding pointer to `nullptr`. Before dereferencing pointers, check if they are `nullptr` and handle the error. In this way, you are protected against accessing a deallocated block.

#### Example - Use of Previously Freed Pointer Error

```
#include <cstdlib>
int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = new int;
    if (pi == NULL) return 0;

    *pi = base_val;
    delete pi;
    j = *pi + shift; //Noncompliant

    return j;
   }
```

The `delete` operator deallocates the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `delete pi;` statement is not valid.

**Correction — Deallocate Pointer After Use**

One possible correction is to deallocate the pointer `pi` only after the last instance where it is accessed.

```
#include <cstdlib>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = new int;
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is deallocated after its last use */
    delete pi;
    return j;
}
```

**Correction — Use `std::unique_ptr`**

Another possible correction is to use a `std::unique_ptr` instead of a raw pointer. Smart pointers such as `std::unique_ptr` manages their own resources. because you don't have to deallocate smart pointers explicitly, they are not inadvertently accessed after deallocation.

```
#include <cstdlib>
#include <memory>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    /* Fix: A smart pointer is used*/
    std::unique_ptr<int>   pi(new int(3));
    if (pi == nullptr) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;
    return j;
}
```

### Pointer or reference to stack variable leaving scope

#### Issue

**Pointer or reference to stack variable leaving scope** occurs when a pointer or reference to a local variable leaves the scope of the variable. For instance:

- A function returns a pointer to a local variable.
- A function performs the assignment `globPtr = &locVar`. `globPtr` is a global pointer variable and `locVar` is a local variable.
- A function performs the assignment `*paramPtr = &locVar`. `paramPtr` is a function parameter that is, for instance, an `int**` pointer and `locVar` is a local `int` variable.
- A C++ method performs the assignment `memPtr = &locVar`. `memPtr` is a pointer data member of the class the method belongs to. `locVar` is a variable local to the method.

The defect also applies to memory allocated using the `alloca` function. The defect does not apply to static, local variables. Polyspace assumes that the local objects within a function definition are in the same scope.

#### Risk

Local variables are allocated an address on the stack. Once the scope of a local variable ends, this address is available for reuse. Using this address to access the local variable value outside the variable scope can cause unexpected behavior.

If a pointer to a local variable leaves the scope of the variable, Polyspace Bug Finder highlights the defect. The defect appears even if you do not use the address stored in the pointer. For maintainable code, it is a good practice to not allow the pointer to leave the variable scope. Even if you do not use the address in the pointer now, someone else using your function can use the address, causing undefined behavior.

#### Fix

Do not allow a pointer or reference to a local variable to leave the variable scope.

#### Example - Pointer to Local Variable Returned from Function

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0; //Noncompliant
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In this example, `func1` returns a pointer to local variable `ret`.

In `main`, `ptr` points to the address of the local variable. When `ptr` is accessed in `func2`, the access is illegal because the scope of `ret` is limited to `func1`,

#### Example - Pointer to Local Variable Escapes Through Lambda Expression

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd; //Noncompliant
  auto adder = [&] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

In this example, the `createAdder` function defines a lambda expression `adder` that captures the local variable `addThis` by reference. The scope of `addThis` is limited to the `createAdder` function. When the object returned by `createAdder` is called, a reference to the variable `addThis` is accessed outside its scope. When accessed in this way, the value of `addThis` is undefined.

#### Correction – Capture Local Variables by Copy in Lambda Expression Instead of Reference

If a function returns a lambda expression object, avoid capturing local variables by reference in the lambda object. Capture the variables by copy instead.

Variables captured by copy have the same lifetime as the lambda object, but variables captured by reference often have a smaller lifetime than the lambda object itself. When the lambda object is used, these variables accessed outside scope have undefined values.

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [=] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

### Accessing object with temporary lifetime

#### Issue

**Accessing object with temporary lifetime** occurs when you attempt to read from or write to an object with temporary lifetime that is returned by a function call. In a structure or union returned by a function, and containing an array, the array members are temporary objects. The lifetime of temporary objects ends:

- When the full expression or full declarator containing the call ends, as defined in the C11 Standard.
- After the next sequence point, as defined in the C90 and C99 Standards. A sequence point is a point in the execution of a program where all previous evaluations are complete and no subsequent evaluation has started yet.

For C++ code, **Accessing object with temporary lifetime** raises a defect only when you write to an object with a temporary lifetime.

If the temporary lifetime object is returned by address, no defect is raised.

#### Risk

Modifying objects with temporary lifetime is undefined behavior and can cause abnormal program termination and portability issues.

#### Fix

Assign the object returned from the function call to a local variable. The content of the temporary lifetime object is copied to the variable. You can now modify it safely.

#### Example - Modifying Temporary Lifetime Object Returned by Function Call

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

/* func_temp() returns a struct value containing
* an array with a temporary lifetime.
*/
int func(void) {

/*Writing to temporary lifetime object is
 undefined behavior
 */
    return ++(func_temp().a[0]); //Noncompliant
}

void main(void) {
    (void)func();
}
```

In this example, `func_temp()` returns by value a structure with an array member `a`. This member has temporary lifetime. Incrementing it is undefined behavior.

#### Correction — Assign Returned Value to Local Variable Before Writing

One possible correction is to assign the return of the call to `func_temp()` to a local variable. The content of the temporary object `a` is copied to the variable, which you can safely increment.

```
 #include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>
```

```
#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

int func(void) {

/* Assign object returned by function call to
 *local variable
 */
    struct S_Array s = func_temp();

/* Local variable can safely be
 *incremented
 */
    ++(s.a[0]);
    return s.a[0];
}

void main(void) {
    (void)func();
}
```

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP54-CPP

# CERT C++: EXP55-CPP

Do not access a cv-qualified object through a cv-unqualified type

## Description

### Rule Definition

*Do not access a cv-qualified object through a cv-unqualified type.*

### Polyspace Implementation

The rule checker checks for **Cast removes cv-qualification of pointer**.

## Examples

### Cast removes cv-qualification of pointer

#### Issue

The issue occurs when a cast removes a `const` or `volatile` qualification from the type of a pointer or reference.

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP55-CPP

# CERT C++: EXP57-CPP

Do not cast or delete pointers to incomplete classes

## Description

### Rule Definition

*Do not cast or delete pointers to incomplete classes.*

### Polyspace Implementation

The rule checker checks for **Conversion or deletion of incomplete class pointer**.

## Examples

### Conversion or deletion of incomplete class pointer

#### Issue

**Conversion or deletion of incomplete class pointer** occurs when you delete or cast to a pointer to an incomplete class. An incomplete class is one whose definition is not visible at the point where the class is used.

For instance, the definition of class `Body` is not visible when the `delete` operator is called on a pointer to `Body`:

```
class Handle {
  class Body *impl;
public:
  ~Handle() { delete impl; }
  // ...
};
```

#### Risk

When you delete a pointer to an incomplete class, it is not possible to call any nontrivial destructor that the class might have. If the destructor performs cleanup activities such as memory deallocation, these activities do not happen.

A similar problem happens, for instance, when you downcast to a pointer to an incomplete class (downcasting is casting from a pointer to a base class to a pointer to a derived class). At the point of downcasting, the relationship between the base and derived class is not known. In particular, if the derived class inherits from multiple classes, at the point of downcasting, this information is not available. The downcasting cannot make the necessary adjustments for multiple inheritance and the resulting pointer cannot be dereferenced.

A similar statement can be made for upcasting (casting from a pointer to derived class to a pointer to a base class).

#### Fix

When you delete or downcast to a pointer to a class, make sure that the class definition is visible.

Alternatively, you can perform one of these actions:

- Instead of a regular pointer, use the `std::shared_ptr` type to point to the incomplete class.
- When downcasting, make sure that the result is valid. Write error-handling code for invalid results.

**Example - Deletion of Pointer to Incomplete Class**

```
class Handle {
  class Body *impl;
public:
  ~Handle() { delete impl; } //Noncompliant
  // ...
};
```

In this example, the definition of class `Body` is not visible when the pointer to `Body` is deleted.

**Correction — Define Class Before Deletion**

One possible correction is to make sure that the class definition is visible when a pointer to the class is deleted.

```
class Handle {
  class Body *impl;
public:
  ~Handle();
  // ...
};

// Elsewhere
class Body { /* ... */ };

Handle::~Handle() {
  delete impl;
}
```

**Correction — Use `std::shared_ptr`**

Another possible correction is to use the `std::shared_ptr` type instead of a regular pointer.

```
#include <memory>

class Handle {
  std::shared_ptr<class Body> impl;
  public:
    Handle();
    ~Handle() {}
    // ...
};
```

**Example - Downcasting to Pointer to Incomplete Class**

`File1.h`:

```
class Base {
protected:
  double var;
public:
  Base() : var(1.0) {}
  virtual void do_something();
  virtual ~Base();
};
```

`File2.h`:

```
void funcprint(class Derived *);
class Base *get_derived();
```

`File1.cpp`:

```
#include "File1.h"
#include "File2.h"

void getandprint() {
  Base *v = get_derived();
  funcprint(reinterpret_cast<class Derived *>(v)); //Noncompliant
}
```

`File2.cpp`:

```
#include "File2.h"
#include "File1.h"
#include <iostream>

class Base2 {
protected:
  short var2;
public:
  Base2() : var2(12) {}
};

class Derived : public Base2, public Base {
  float var_derived;
public:
    Derived() : Base2(), Base(), var_derived(1.2f) {}
    void do_something()
    {
        std::cout << "var_derived: "
                  << var_derived << ", var : " << var
                  << ", var2: " << var2 << std::endl;
    }
 };

void funcprint(Derived *d) {
  d->do_something();
}

Base *get_derived() {
  return new Derived;
}
```

In this example, the definition of class `Derived` is not visible in `File1.cpp` when a `Base*` pointer to downcast to a `Derived*` pointer.

In `File2.cpp`, class `Derived` derives from two classes, `Base` and `Base2`. This information about multiple inheritance is not available at the point of downcasting in `File1.cpp`. The result of downcasting is passed to the function `funcprint` and dereferenced in the body of `funcprint`. Because the downcasting was done with incomplete information, the dereference can be invalid.

#### Correction — Define Class Before Downcasting

One possible correction is to define the class `Derived` before downcasting a `Base*` pointer to a `Derived*` pointer.

In this corrected example, the downcasting is done in `File2.cpp` in the body of `funcprint` at a point where the definition of class `Derived` is visible. The downcasting is not done in `File1.cpp` where the definition of `Derived` is not visible. The changes from the previous incorrect example are highlighted.

`File1_corr.h`:

```
class Base {
protected:
  double var;
public:
  Base() : var(1.0) {}
  virtual void do_something();
  virtual ~Base();
};
```

`File2_corr.h`:

```
void funcprint(class Base *);
class Base *get_derived();
```

`File1.cpp`:

```
#include "File1_corr.h"
#include "File2_corr.h"

void getandprint() {
  Base *v = get_derived();
  funcprint(v);
}
```

`File2.cpp`:

```
#include "File2_corr.h"
#include "File1_corr.h"
#include <iostream>

class Base2 {
protected:
  short var2;
public:
  Base2() : var2(12) {}
};

class Derived : public Base2, public Base {
  float var_derived;

public:
    Derived() : Base2(), Base(), var_derived(1.2f) {}
    void do_something()
    {
        std::cout << "var_derived: "
                  << var_derived << ", var : " << var
                  << ", var2: " << var2 << std::endl;
    }
};

void funcprint(Base *d) {
  Derived *temp = dynamic_cast<Derived*>(d);
  if(temp)  {
     d->do_something();
  }
  else {
      //Handle error
  }
}

Base *get_derived() {
  return new Derived;
}
```

### Check Information

**Group:** 02. Expressions (EXP)

## Version History

**Introduced in R2019a**

### See Also

Check SEI CERT-C++ (-cert-cpp))

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

EXP57-CPP

# CERT C++: EXP58-CPP

Pass an object of the correct type to va_start

## Description

### Rule Definition

*Pass an object of the correct type to va_start.*

### Polyspace Implementation

The rule checker checks for **Incorrect type data passed to va_start**.

## Examples

### Incorrect type data passed to va_start

#### Issue

**Incorrect type data passed to va_start** occurs when the second argument of the `va_start` macro has one of these data types:

- A data type that changes when undergoing default argument promotion.

  For instance, `char` and `short` undergo promotion to `int` or `unsigned int` and `float` undergoes promotion to `double`. The types `int` and `double` do not change under default argument promotion.
- (C only) A register type or a data type declared with the `register` qualifier.
- (C++ only) A reference data type.
- (C++ only) A data type that has a nontrivial copy constructor or a nontrivial move constructor.

#### Risk

In a variadic function or function with variable number of arguments:

```
void multipleArgumentFunction(int someArg, short rightmostFixedArg, ...) {
    va_list myList;
    va_start(myList, rightmostFixedArg);
    ...
    va_end(myList);
}
```

The `va_start` macro initializes a variable argument list so that additional arguments to the variadic function after the fixed parameters can be captured in the list. According to the C11 and C++14 Standards, if you use one of the flagged data types for the second argument of the `va_start` macro (for instance, `rightmostFixedArg` in the preceding example), the behavior is undefined.

If the data type involves a nontrivial copy constructor, the behavior is implementation-defined. For instance, whether the copy constructor is invoked in the call to `va_start` depends on the compiler.

#### Fix

When using the `va_start` macro, try to use the types `int`, `unsigned int` or `double` for the rightmost named parameter of the variadic function. Then, use this parameter as the second argument of the `va_start` macro.

For instance, in this example, the rightmost named parameter of the variadic function has a supported data type `int`:

```
void multipleArgumentFunction(int someArg, int rightmostFixedArg, ...) {
    va_list myList;
    va_start(myList, rightmostFixedArg);
    ...
    va_end(myList);
}
```

To avoid undefined and implementation-defined behavior, minimize the use of variadic functions. Use the checkers for `MISRA C:2012 Rule 17.1` or `MISRA C++:2008 Rule 8-4-1` to detect use of variadic functions.

#### Example – Incorrect Data Types for Second Argument of va_start

```
#include <string>
#include <cstdarg>

double addVariableNumberOfDoubles(double* weight, short num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num); //Noncompliant
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}

double addVariableNumberOfFloats(float* weight, int num, std::string s, ...) {
    float sum=0.0;
    va_list list;
    va_start(list, s); //Noncompliant
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, float);
    }
    va_end(list);
    return sum;
}
```

In this example, the checker flags the call to `va_start` in:

- `addVariableNumberOfDoubles` because the argument has type `short`, which undergoes default argument promotion to `int`.
- `addVariableNumberOfFloats` because the argument has type `std::string`, which has a nontrivial copy constructor.

#### Correction — Fix Data Type for Second Argument of va_start

Make sure that the second argument of the `va_start` macro has a supported data type. In the following corrected example:

- In `addVariableNumberOfDoubles`, the data type of the last named parameter of the variadic function is changed to `int`.
- In `addVariableNumberOfFloats`, the second and third parameters of the variadic function are switched so that data type of the last named parameter is `int`.

```
#include <string>
#include <cstdarg>

double addVariableNumberOfDoubles(double* weight, int num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}

double addVariableNumberOfFloats(double* weight, std::string s, int num, ...) {
    double sum=0.0;
    va_list list;
    va_start(list, num);
    for(int i=0; i < num; i++) {
        sum+=weight[i]*va_arg(list, double);
    }
    va_end(list);
    return sum;
}
```

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP58-CPP

# CERT C++: EXP59-CPP

Use offsetof() on valid types and members

## Description

### Rule Definition

*Use offsetof() on valid types and members.*

### Polyspace Implementation

The rule checker checks for **Incorrect use of offsetof in C++**.

## Examples

### Incorrect use of offsetof in C++

#### Issue

This defect occurs when you pass arguments to the `offsetof` macro for which the behavior of the macro is not defined.

The `offsetof` macro:

```
offsetof(classType, aMember)
```

returns the offset in bytes of the data member `aMember` from the beginning of an object of type `classType`. For use in `offsetof`, `classType` and `aMember` have certain restrictions:

- `classType` must be a standard layout class.

  For instance, it must not have `virtual` member functions. For more information on the requirements for a standard layout class, see C++ named requirements: StandardLayoutType.
- `aMember` must not be static.
- `aMember` must not be a member function.

The checker flags uses of the `offsetof` macro where the arguments violate one or more of these restrictions.

#### Risk

Violating the restrictions on the arguments of the `offsetof` macro leads to undefined behavior.

#### Fix

Use the `offsetof` macro only on nonstatic data members of a standard layout class.

The result details state which restriction on the `offsetof` macro is violated. Fix the violation.

#### Example – Use of `offsetof` Macro with Nonstandard Layout Class

```
#include <cstddef>
```

```
class myClass {
     int privateData;
  public:
     int publicData;
};

void func() {
  size_t off = offsetof(myClass, publicData); //Noncompliant
  // ...
}
```

In this example, the class `myClass` has two data members with different access control, one private and the other public. Therefore, the class does not satisfy the requirements of a standard layout class and cannot be used with the `offsetof` macro.

**Correction — Use Uniform Access Control for All Data Members**

If the use of `offsetof` is important for the application, make sure that the first argument is a class with a standard layout. For instance, see if you can work around the need for a public data member.

```
#include <cstddef>

class myClass {
     int member1;
     int member2;
  public:
     int getMember2(void) { return member2;}
     friend void func(void);
};

void func() {
  size_t off = offsetof(myClass, member2);
  // ...
}
```

## Check Information

**Group:** 02. Expressions (EXP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

EXP59-CPP

# CERT C++: EXP61-CPP

A lambda object must not outlive any of its reference captured objects

## Description

### Rule Definition

*A lambda object must not outlive any of its reference captured objects.*

### Polyspace Implementation

The rule checker checks for **Object Escapes Scope Through Lambda Expression**.

## Examples

### Object Escapes Scope Through Lambda Expression

#### Issue

The issue occurs when a lambda expression captures an object *by reference* and the lambda expression object outlives the captured object. For instance, the captured object is a local variable but the lambda expression object has a much larger scope.

#### Risk

If a lambda expression object outlives one of its reference captured objects, the captured object can be accessed outside its scope.

For instance, consider this function `createFunction`:

```
std::function<std::int32_t()> createFunction() {
   std::int32_t localVar = 0;
   return ([&localVar]() -> std::int32_t {
       localVar = 1;
       return localVar;
   });
}
```

`createFunction` returns a lambda expression object that captures the local variable `localVar` *by reference*. The scope of `localVar` is limited to `createFunction` but the lambda expression object returned has a much larger scope.

This situation can result in an attempt to access the local object `localVar` outside its scope. For instance, when you call `createFunction` and assign the returned lambda expression object to another object `aFunction`:

```
auto aFunction = createFunction();
```

and then invoke the new object `aFunction`:

```
std::int32_t someValue = aFunction();
```

the captured variable `localVar` is no longer in scope. Therefore, the value returned from `aFunction` is undefined.

#### Fix

If a function returns a lambda expression, to avoid accessing a captured object outside its scope, make sure that the lambda expression captures all objects by copy. For instance, you can rewrite `createFunction` as:

```
std::function<std::int32_t()> createFunction() {
   std::int32_t localVar = 0;
   return ([localVar]() mutable -> std::int32_t {
       localVar = 1;
       return localVar;
   });
}
```

#### Example – Pointer to Local Variable Escapes Through Lambda Expression

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd; //Noncompliant
  auto adder = [&] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

In this example, the `createAdder` function defines a lambda expression `adder` that captures the local variable `addThis` by reference. The scope of `addThis` is limited to the `createAdder` function. When the object returned by `createAdder` is called, a reference to the variable `addThis` is accessed outside its scope. When accessed in this way, the value of `addThis` is undefined.

#### Correction – Capture Local Variables by Copy in Lambda Expression Instead of Reference

If a function returns a lambda expression object, avoid capturing local variables by reference in the lambda object. Capture the variables by copy instead.

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [=] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

## Check Information

**Group:** 02. Expressions (EXP)

## Version History

**Introduced in R2019b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

EXP61-CPP

# CERT C++: EXP62-CPP

Do not access the bits of an object representation that are not part of the object's value representation

## Description

### Rule Definition

*Do not access the bits of an object representation that are not part of the object's value representation.*

### Polyspace Implementation

The rule checker checks for **Accessing padding and `vtable` bits**.

## Examples

### Accessing Padding and `vtable` Bits

#### Issue

This issue occurs when either of these conditions is true:

- You use the function `std::memcmp` on padding bits. For example:

  ```
  class A{char str; int size;}; //Padding beteen the fields.
  void foo(){
      A a1,a2;
      if(std::memcmp(&a1, &a2, sizeof(A))){/**/}//Noncompliant
  }
  ```
- You use the functions `memset` or `memcmp` on `vtable` bits. For example:

  ```
  class A{
  public:
      void virtual eval(){}
  }//Class contains a vtable
  void foo(){
      A *a = new A;
      //...
      std::memset(a,0,sizeof(A));//Noncompliant
      //...
      a->eval();
  }
  ```

The `vtable` holds pointers to different virtual functions available to an object. Padding bits might be inserted between members of an object. These bits are not part of the value representation of an object.

As an exception, this rule allows accessing `vtable` and padding bits if the code does not make assumptions about their values. This rule also allows overwriting the padding bits. For instance, you can use `memcpy()` to copy all object bits containing a bitfield into another object. For instance:

```
struct BF {
    int i : 10;
```

```
    int j;
};
void foo(const BF &a1)
{
    BF a2;
    std::memcpy(&a2, &a1, sizeof(BF));//Compliant
}
```

Depending on your environment and usage, unions might not have any padding bits. Violation of this rule is not reported on unions.

#### Risk

The C++ standard does not specify how the padding bit values and the `vtable` values are set. The values in these bits can vary depending on your hardware and software. Relying on such implementation-dependent values in your code might result in unexpected results that are difficult to diagnose.

#### Fix

Avoid accessing padding bits and `vtable` bits. In C++, the best practice is to avoid functions that directly access object bits, such as `std::memcmp()` and `std::memset()`.

#### Example — Avoid Comparing Padding Bits

```
#include <cstring>
class myString  {
    unsigned char buffer;
public:
    int size;
    void set(const char* );
    char* get();
};

void compmyString(const myString &s1, const myString &s2)
{
    if (!std::memcmp(&s1, &s2, sizeof(myString))) { //Noncompliant
        // ...
    }
}
```

This example implements a string type class `mystring`. The function `compmyString` compares two objects of this class. The comparison is performed by calling `std::memcmp()` to compare the bits of two objects, including the padding bits. Because the values of the padding bits are implementation-defined, using `std::memcmp()` makes the code nonportable and might yield unexpected results. Polyspace reports a violation of this rule.

#### Correction — Compare Value Representation of Objects

When you compare objects, the best practice is to implement appropriately overloaded operators that perform a field-by-field comparison. For instance, to check if two `myString` objects are equal, this code implements the `operator==`, which compares the fields `buffer` and `size`. Using the `operator==` avoids padding bits comparison.

```
#include <cstring>
class myString  {
    unsigned char buffer;
public:
    int size;
    void set(const char* );
    char* get();
    friend bool operator==(const myString &left, const myString &right)
    {
        return left.buffer == right.buffer &&
        left.size == right.size;
    }
};

void compmyString(const myString &s1, const myString &s2)
{
    if (s1==s2) { //Compliant
        // ...
    }
}
```

### Check Information

**Group:** 02. Expressions (EXP)

## Version History

**Introduced in R2022b**

### See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

EXP62-CPP

# CERT C++: EXP63-CPP

Do not rely on the value of a moved-from object

## Description

### Rule Definition

*Do not rely on the value of a moved-from object.*[10]

### Polyspace Implementation

The rule checker checks for **Reading the Value of a Moved-from Object**.

## Examples

### Reading the Value of a Moved-from Object

#### Issue

This issue occurs when the value of a source object is read after its content is moved to a destination object by calling the `std::move` function explicitly. Polyspace does not flag accessing the value of a moved-from object if:

- The source object of an explicit move operation is of these types:
  - `std::unique_ptr`
  - `std::shared_ptr`
  - `std::weak_ptr`
  - `std::basic_ios`
  - `std::basic_filebuf`
  - `std::thread`
  - `std::unique_lock`
  - `std::shared_lock`
  - `std::promise`

[10] *This software has been created by MathWorks incorporating portions of: the "SEI CERT-C Website," © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, "SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition," © 2016 Carnegie Mellon University, and "SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition" © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

  - `std::future`
  - `std::shared_future`
  - `std::packaged_task`

  These objects do not remain in an unspecified state after their state is moved explicitly.
- The move operation is performed implicitly. For instance, the function `std::remove` might access the state of a source object after an implicit move operation. Polyspace does not flag it. A best practice is to avoid such operations and use safer alternatives that prevent accidental access, such as `std::erase`.
- The source object is of a built-in base type, such as: `int`, `enum`, `float`, `double`, pointer, `std::intptr_t`, `std::nullptr_t`.

**Risk**

Because the state of a source object is generally unspecified after a move operation, it is unsafe to perform operations that rely on the state of the source object after a move operation. Accessing the state of the source object after a move operation might result in a data integrity violation, an unexpected value, or an illegal dereferencing of a pointer.

**Fix**

Avoid operations that might read a source object after its content is moved.

**Example — Reading Value of Source Object After Calling `std::move`**

```
#include<string>
#include<iostream>
void F1()
{
    std::string s1{"string"};
    std::string s2{std::move(s1)};
    // ...
    std::cout
    <<  // Noncompliant
    s1
    << "\n";
    // value after move operation
}
void g(std::string v)
{
    std::cout << v << std::endl;
}

void F3()
{
    std::string s;
    for (unsigned i = 0; i < 10; ++i) {
        s.append(1, static_cast<char>('0' + i));  //Noncompliant
        g(std::move(s));
    }
}
```

- In the function `F1`, the string `s1` is explicitly moved to `s2` by calling `std::move`. After the move operation, the function attempts to read `s1`. Polyspace flags this attempt of reading a source object after an explicit move.

- In the function F3, the string s is explicitly moved and then it is read by the std::string::append function. Polyspace flags this attempt of reading a source object after an explicit move.

**Correction — Read Values of Source Objects in Specified State**

```
#include<string>
#include<iostream>
void F2()
{
    std::unique_ptr<std::int32_t> ptr1 = std::make_unique<std::int32_t>(0);
    std::unique_ptr<std::int32_t> ptr2{std::move(ptr1)};
    std::cout << ptr1.get() << std::endl; // Compliant by exception
}
void g(std::string v)
{
    std::cout << v << std::endl;
}
void F4()
{
    for (unsigned i = 0; i < 10; ++i) {
        std::string s(1, static_cast<char>('0' + i)); // Compliant
        g(std::move(s));
    }
}
```

- In the function F2, the unique pointer ptr1 is explicitly moved to ptr2. Because the state of std::unique_ptr remains in a specified state after the move, reading a source unique pointer after an explicit move is compliant.
- In the function F4, the string s is explicitly moved. In each iteration of the loop, s is initiated to specific content before the move operation is triggered. As a result, the state of s is specified before the object is accessed. This method of accessing the source object after a move operation is compliant with this rule.

## Check Information

**Group:** Rule 02. Expressions (EXP)

# Version History

**Introduced in R2021a**

## See Also

Check SEI CERT-C++ (-cert-cpp)) | AUTOSAR C++14 Rule A12-8-3

**Topics**

“Check for and Review Coding Standard Violations”

**External Websites**

EXP63-CPP

# CERT C++: INT30-C

Ensure that unsigned integer operations do not wrap

## Description

### Rule Definition

*Ensure that unsigned integer operations do not wrap.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Unsigned integer overflow**.
- **Unsigned integer constant overflow**.

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

### Extend Checker

When the input values are unknown and only a subset of inputs cause an issue, Bug Finder might not detect an **Unsigned integer overflow** or **Unsigned integer constant overflow**. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Unsigned integer overflow

#### Issue

**Unsigned integer overflow** occurs when an operation on unsigned integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

The C11 standard states that unsigned integer overflows result in wrap-around behavior. However, a wrap around behavior might not always be desirable. For instance, if the result of a computation is used as an array size and the computation overflows, the array size is much smaller than expected.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do

not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling. In the error handling code, you can override the default wrap-around behavior for overflows and implement saturation behavior, for instance.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Add One to Maximum Unsigned Integer

```
#include <limits.h>

unsigned int plusplus(void) {

    unsigned uvar = UINT_MAX;
    uvar++; //Noncompliant
    return uvar;
}
```

In the third statement of this function, the variable `uvar` is increased by 1. However, the value of `uvar` is the maximum unsigned integer value, so 1 plus the maximum integer value cannot be represented by an `unsigned int`. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `uvar` is reduced by modulo `UINT_MAX`. The result is `uvar = 1`.

#### Correction — Different Storage Type

One possible correction is to store the operation result in a larger data type. In this example, by returning an `unsigned long long` instead of an `unsigned int`, the overflow error is fixed.

```
#include <limits.h>

unsigned long long plusplus(void) {

    unsigned long long ullvar = UINT_MAX;
    ullvar++;
    return ullvar;
}
```

### Unsigned integer constant overflow

#### Issue

**Unsigned integer constant overflow** occurs when you assign a compile-time constant to a unsigned integer variable whose data type cannot accommodate the value. An `n`-bit unsigned integer holds values in the range `[0, 2^n-1]`.

For instance, `c` is an 8-bit unsigned `char` variable that cannot hold the value 256.

```
unsigned char c = 256;
```

To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.

#### Risk

The C standard states that overflowing unsigned integers must be wrapped around (see, for instance, the C11 standard, section 6.2.5). However, the wrap-around behavior can be unintended and cause unexpected results.

#### Fix

Check if the constant value is what you intended. If the value is correct, use a wider data type for the variable.

#### Example - Overflowing Constant from Macro Expansion

```
#define MAX_UNSIGNED_CHAR 255 //Noncompliant
#define MAX_UNSIGNED_SHORT 65535 //Noncompliant

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned short c2 = MAX_UNSIGNED_SHORT + 1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow.

#### Correction — Use Wider Data Type

One possible correction is to use a wider data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned short c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned int c2 = MAX_UNSIGNED_SHORT + 1;
}
```

## Check Information

**Group:** 03. Integers (INT)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

INT30-C

# CERT C++: INT31-C

Ensure that integer conversions do not result in lost or misinterpreted data

## Description

### Rule Definition

*Ensure that integer conversions do not result in lost or misinterpreted data.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Integer conversion overflow**.
- **Call to memset with unintended value**.
- **Sign change integer conversion overflow**.
- **Tainted sign change conversion**.
- **Unsigned integer conversion overflow**.

### Extend Checker

Extend this checker to run a stricter analysis:

- When the input values are unknown and only a subset of inputs causes an issue, a default Bug Finder analysis might not detect an **Integer conversion overflow**, **Sign change integer conversion overflow**, or **Unsigned integer conversion overflow**. To check for these issues caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.
- A default Bug Finder analysis might not flag a **Tainted sign change conversion** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Integer conversion overflow

#### Issue

**Integer conversion overflow** occurs when converting an integer to a smaller integer type. If the variable does not have enough bytes to represent the original value, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer conversion overflows result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface".

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Converting from `int` to `char`

```
char convert(void) {

    int num = 1000000;

    return (char)num; //Noncompliant
}
```

In the return statement, the integer variable `num` is converted to a `char`. However, an 8-bit or 16-bit character cannot represent 1000000 because it requires at least 20 bits. So the conversion operation overflows.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number.

```
long convert(void) {

    int num = 1000000;

    return (long)num;
}
```

### Call to memset with unintended value

#### Issue

**Call to memset with unintended value** occurs when Polyspace Bug Finder detects a use of the `memset` or `wmemset` function with possibly incorrect arguments.

`void *memset (void *ptr, int value, size_t num)` fills the first `num` bytes of the memory block that `ptr` points to with the specified `value`. If the argument `value` is incorrect, the memory block is initialized with an unintended value.

The unintended initialization can occur in the following cases.

| Issue | Risk | Possible Fix |
|---|---|---|
| The second argument is `'0'` instead of `0` or `'\0'`. | The ASCII value of character `'0'` is `48` (decimal), `0x30` (hexadecimal), `069` (octal) but not `0` (or `'\0'`) . | If you want to initialize with `'0'`, use one of the ASCII values. Otherwise, use `0` or `'\0'`. |
| The second and third arguments are probably reversed. For instance, the third argument is a literal and the second argument is not a literal. | If the order is reversed, a memory block of unintended size is initialized with incorrect arguments. | Reverse the order of the arguments. |
| The second argument cannot be represented in a byte. | If the second argument cannot be represented in a byte, and you expect each byte of a memory block to be filled with that argument, the initialization does not occur as intended. | Apply a bit mask to the argument to produce a wrapped or truncated result that can be represented in a byte. When you apply a bit mask, make sure that it produces an expected result.<br>For instance, replace `memset(a, -13, sizeof(a))` with `memset(a, (-13) & 0xFF, sizeof(a))`. |

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Value Cannot Be Represented in a Byte

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE];
    int c = -2;
    memset(buf, (char)c, sizeof(buf)); //Noncompliant
}
```

In this example, `(char)c` cannot be represented in a byte.

#### Correction — Apply Bit Mask

One possible correction is to apply a bit mask so that the result can be represented in a byte. However, check that the result is an acceptable initialization value.

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE   ];
    int c = -2;
    memset(buf, c & 0xFF, sizeof(buf));
}
```

### Sign change integer conversion overflow

#### Issue

**Sign change integer conversion overflow** occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Convert from `unsigned char` to `char`

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count; //Noncompliant
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

#### Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

### Tainted sign change conversion

#### Issue

**Tainted sign change conversion** looks for values from unsecure sources that are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
memmove
strncmp
strncpy
calloc
malloc
memalign
```

#### Risk

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

#### Fix

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

#### Example - Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(int size) {
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size); //Noncompliant
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

#### Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(int size) {
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

### Unsigned integer conversion overflow

#### Issue

**Unsigned integer conversion overflow** occurs when converting an unsigned integer to a smaller unsigned integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer conversion overflows result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.

- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Converting from `int` to `char`

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum; //Noncompliant
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned long)unum;
}
```

## Check Information

**Group:** 03. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

INT31-C

# CERT C++: INT32-C

Ensure that operations on signed integers do not result in overflow

## Description

### Rule Definition

*Ensure that operations on signed integers do not result in overflow.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Integer overflow**.
- **Tainted division operand**.
- **Tainted modulo operand**.

### Extend Checker

Extend this checker to run a stricter analysis:

- When the input values are unknown and only a subset of inputs causes an issue, a default Bug Finder analysis might not detect an **Integer overflow**. To check for this issue when it is caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.
- A default Bug Finder analysis might not flag a **Tainted division operand** or **Tainted modulo operand** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Integer overflow

#### Issue

**Integer overflow** occurs when an operation on integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer overflows on signed integers result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the overflowing computation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++; //Noncompliant
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

#### Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

### Tainted division operand

#### Issue

**Tainted division operand** detects division operations where one or both of the integer operands is from an unsecure source.

#### Risk

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

#### Fix

Before performing the division, validate the values of the operands. Check for denominators of `0` or `-1`, and numerators of the minimum integer value.

#### Example - Division of Function Arguments

```
#include <limits.h>
#include <stdio.h>

extern void print_int(int);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    int r =  num/denum; //Noncompliant
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

#### Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include <limits.h>
#include <stdio.h>

extern void print_long(long);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    long res= 0;
    if (denum!=0 && !(num==INT_MIN && denum==-1)) {
        res = num/denum;
    }
    print_long(res);
    return res;
}
```

### Tainted modulo operand

#### Issue

**Tainted modulo operand** checks the operands of remainder `%` operations. Bug Finder flags modulo operations with one or more tainted operands.

#### Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

#### Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of `0` and `-1`. Check both operands for negative values.

#### Example - Modulo with Function Arguments

```
#include <stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem =  128%userden; //Noncompliant
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using an input argument. The argument is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

#### Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
#include<stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem = 0;
    if (userden > 0 ) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

## Check Information

**Group:** 03. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

INT32-C

# CERT C++: INT33-C

Ensure that division and remainder operations do not result in divide-by-zero errors

## Description

### Rule Definition

*Ensure that division and remainder operations do not result in divide-by-zero errors.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Integer division by zero**.
- **Tainted division operand**.
- **Tainted modulo operand**.

### Extend Checker

Extend this checker to run a stricter analysis:

- When the input values are unknown and only a subset of inputs causes an issue, a default Bug Finder analysis might not detect an **Integer division by zero**. To check for this issue when it is caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".
- A default Bug Finder analysis might not flag a **Tainted division operand** or **Tainted modulo operand** issue for certain inputs that originate outside of the current analysis boundary. See "Sources of Tainting in a Polyspace Analysis". To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Integer division by zero

#### Issue

**Integer division by zero** occurs when the denominator of a division or modulo operation can be a zero-valued integer.

#### Risk

A division by zero can result in a program crash.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the denominator variable acquires a zero value. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface".

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Example - Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    result = num/denom; //Noncompliant

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

### Correction — Check Before Division

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

### Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2;
    int result = 0;
    result = num/denom;

    return result;
}
```

#### Example - Modulo Operation with Zero

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        arr[i] = input % i; //Noncompliant
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

In this example, Polyspace flags the modulo operation as a division by zero. Because modulo is inherently a division operation, the divisor (right hand argument) cannot be zero. The modulo operation uses the `for` loop index as the divisor. However, the `for` loop starts at zero, which cannot be an iterator.

#### Correction — Check Divisor Before Operation

One possible correction is checking the divisor before the modulo operation. In this example, see if the index `i` is zero before the modulo operation.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        if(i != 0)
        {
             arr[i] = input % i;
        }
        else
        {
             arr[i] = input;
        }
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

#### Correction — Change Divisor

Another possible correction is changing the divisor to a nonzero integer. In this example, add one to the index before the `%` operation to avoid dividing by zero.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
         arr[i] = input % (i+1);
    }
    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

### Tainted division operand

#### Issue

**Tainted division operand** detects division operations where one or both of the integer operands is from an unsecure source.

#### Risk

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

#### Fix

Before performing the division, validate the values of the operands. Check for denominators of `0` or `-1`, and numerators of the minimum integer value.

#### Example - Division of Function Arguments

```
#include <limits.h>
#include <stdio.h>

extern void print_int(int);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    int r =  num/denum; //Noncompliant
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

#### Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include <limits.h>
#include <stdio.h>

extern void print_long(long);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    long res= 0;
    if (denum!=0 && !(num==INT_MIN && denum==-1)) {
        res = num/denum;
    }
    print_long(res);
    return res;
}
```

### Tainted modulo operand

#### Issue

**Tainted modulo operand** checks the operands of remainder % operations. Bug Finder flags modulo operations with one or more tainted operands.

#### Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is -1, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

#### Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of 0 and -1. Check both operands for negative values.

#### Example - Modulo with Function Arguments

```
#include <stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem =  128%userden; //Noncompliant
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using an input argument. The argument is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

#### Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
#include<stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem = 0;
    if (userden > 0 ) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

## Check Information

**Group:** 03. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

INT33-C

# CERT C++: INT34-C

Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand

## Description

### Rule Definition

*Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Shift of a negative value**.
- **Shift operation overflow**.

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

### Extend Checker

When the input values are unknown and only a subset of inputs cause an issue, Bug Finder might not detect a **Shift of a negative value** or **Shift operation overflow**. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Shift of a negative value

#### Issue

**Shift of a negative value** occurs when a bit-wise shift is used on a variable that can have negative values.

#### Risk

Shifts on negative values overwrite the sign bit that identifies a number as negative. The shift operation can result in unexpected values.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being shifted acquires negative values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

To fix the defect, check for negative values before the bit-wise shift operation and perform appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Shifting a negative variable

```
int shifting(int val)
{
    int res = -1;
    return res << val; //Noncompliant
}
```

In the return statement, the variable `res` is shifted a certain number of bits to the left. However, because `res` is negative, the shift might overwrite the sign bit.

#### Correction — Change the Data Type

One possible correction is to change the data type of the shifted variable to unsigned. This correction eliminates the sign bit, so left shifting does not change the sign of the variable.

```
int shifting(int val)
{
    unsigned int res = -1;
    return res << val;
}
```

### Shift operation overflow

#### Issue

**Shift operation overflow** occurs when a shift operation can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Shift operation overflows can result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variables in the shift operation acquire their current values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface".

You can fix the defect by:

- Using a bigger data type for the result of the shift operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Left Shift of Integer

```
int left_shift(void) {

    int foo = 33;
    return 1 << foo; //Noncompliant
}
```

In the return statement of this function, bit-wise shift operation is performed shifting 1 `foo` bits to the left. However, an `int` has only 32 bits, so the range of the shift must be between 0 and 31. Therefore, this shift operation causes an overflow.

#### Correction — Different storage type

One possible correction is to store the shift operation result in a larger data type. In this example, by returning a `long long` instead of an `int`, the overflow defect is fixed.

```
long long left_shift(void) {

    int foo = 33;
    return 1LL << foo;
}
```

## Check Information

**Group:** 03. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

INT34-C

# CERT C++: INT35-C

Use correct integer precisions

## Description

### Rule Definition

*Use correct integer precisions.*

### Polyspace Implementation

The rule checker checks for **Integer precision exceeded**.

## Examples

### Integer precision exceeded

#### Issue

**Integer precision exceeded** occurs when an integer expression uses the integer size in an operation that exceeds the integer precision. On some architectures, the size of an integer in memory can include sign and padding bits. On these architectures, the integer size is larger than the precision which is just the number of bits that represent the value of the integer.

#### Risk

Using the size of an integer in an operation on the integer precision can result in integer overflow, wrap around, or unexpected results. For instance, an unsigned integer can be stored in memory in 64 bits, but uses only 48 bits to represent its value. A 56 bits left-shift operation on this integer is undefined behavior.

Assuming that the size of an integer is equal to its precision can also result in program portability issues between different architectures.

#### Fix

Do not use the size of an integer instead of its precision. To determine the integer precision, implement a precision computation routine or use a builtin function such as `__builtin_popcount()`.

#### Example - Using Size of `unsigned int` for Left Shift Operation

```
#include <limits.h>

unsigned int func(unsigned int exp)
{
    if (exp >= sizeof(unsigned int) * CHAR_BIT) {
        /* Handle error */
    }
    return 1U << exp; //Noncompliant
}
```

In this example, the function uses a left shift operation to return the value of 2 raised to the power of `exp`. The operation shifts the bits of `1U` by `exp` positions to the left. The `if` statement ensures that the operation does not shift the bits by a number of positions `exp` greater than the size of an `unsigned int`. However, if `unsigned int` contains padding bits, the value returned by `sizeof()` is larger than the precision of `unsigned int`. As a result, some values of `exp` might be too large, and the shift operation might be undefined behavior.

#### Correction — Implement Function to Compute Precision of `unsigned int`

One possible correction is to implement a function `popcount()` that computes the precision of `unsigned int` by counting the number of set bits.

```
#include <stddef.h>
#include <stdint.h>
#include <limits.h>

size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)


unsigned int func(unsigned int exp)
{
    if (exp >= PRECISION(UINT_MAX)) {
        /* Handle error */
    }
    return 1 << exp;
}




size_t popcount(uintmax_t num)
{
    size_t precision = 0;
    while (num != 0) {
        if (num % 2 == 1) {
            precision++;
        }
        num >>= 1;
    }
    return precision;
}
```

## Check Information

**Group:** 03. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

INT35-C

# CERT C++: INT36-C

Converting a pointer to integer or integer to pointer

## Description

### Rule Definition

*Converting a pointer to integer or integer to pointer.*

### Polyspace Implementation

The rule checker checks for **Conversion between pointer and integer**.

## Examples

### Conversion between pointer and integer

#### Issue

**Conversion between pointer and integer** is raised when any of these conditions is true:

- A pointer type is converted into an integral type with smaller size. When converting a pointer of the types `intptr_t` or `uintprt_t` into integral types `enum`, `ptrdiff_t`, or `pid_t`, Polyspace raises a defect if the size of pointers and integer types are different in your environment. In `i386` environment, both pointers and integer types have a size of 32 bits. In this environment, Polyspace does not flag conversion from a pointer to an integer of same size. But in `x86_64` environment where pointers are 64 bits and unsigned integers are 32 bits, Polyspace flags conversion from pointers to integers of different sizes.
- An integral type is converted into a raw pointer type. Polyspace flags conversion from integral types to raw pointers regardless of their sizes. For instance, if an integer originates from a pointer and then later, is cast into a pointer, Polyspace flags the conversion from integer to pointer. In this case, the integer and the pointer are of the same size, but the conversion is still flagged.

#### Risk

The mapping between pointers and integers is not always consistent with the addressing structure of the environment.

Converting from pointers to integers can create:

- Truncated or out of range integer values.
- Invalid integer types.

Converting from integers to pointers can create:

- Misaligned pointers or misaligned objects.
- Invalid pointer addresses.

#### Fix

Where possible, avoid pointer-to-integer or integer-to-pointer conversions. If you want to convert a `void` pointer to an integer, so that you do not change the value, use types:

- C99 — `intptr_t` or `uintptr_t`
- C90 — `size_t` or `ssize_t`

#### Example — Integer to Pointer Conversions

```
unsigned int* badintptrcast(void)
{
    int* ptr;
    unsigned long long int_same_as_ptr ;
    unsigned int int_smaller_than_ptr;

    unsigned int* ptr0 = (unsigned int*)0xdeadbeef; //Noncompliant
    /*int to ptr of same size*/
    ptr = (int*)int_same_as_ptr;  //Noncompliant

    /*int to ptr of different size*/
    ptr = (int*)int_smaller_than_ptr;  //Noncompliant

    return (unsigned int*)(ptr0 - (unsigned int*)ptr);  //Noncompliant
}
```

In this example, Polyspace flags the conversions that might be unsafe. For instance:

- The conversion of `0xdeadbeef` to `unsigned int*` causes alignment issues for the pointer. Polyspace flags the conversion.
- The conversion of the `int_same_as_ptr` and `int_smaller_than_ptr` to `int*` pointers might result in invalid pointer address. Polyspace flags these conversion.
- The `return` statement casts `ptrdiff_t` to a pointer. This pointer might not point to an invalid address. Polyspace flags the conversion.

#### Correction — Use `intptr_t`

One possible correction is to use `intptr_t` types to store addresses. When storing the result of subtracting two pointers, use the type `ptrdiff_t`.

```
#include<cstdint>
#include<stddef.h>
ptrdiff_t badintptrcast(void)
{
    intptr_t ptr;
    unsigned long long int_same_as_ptr ;
    unsigned int int_smaller_than_ptr;

    intptr_t ptr0 = (intptr_t)0xdeadbeef; //Compliant
    /*int to ptr of same size*/
    ptr = (intptr_t)int_same_as_ptr;  //Compliant

    /*int to ptr of different size*/
    ptr = (intptr_t)int_smaller_than_ptr;  //Compliant

    int offset = 0;
    return (ptrdiff_t)(ptr0 - offset);//Compliant
}
```

## Check Information

**Group:** 03. Integers (INT)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

INT36-C

# CERT C++: ARR30-C

Do not form or use out-of-bounds pointers or array subscripts

## Description

### Rule Definition

*Do not form or use out-of-bounds pointers or array subscripts.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access out of bounds**.
- **Pointer access out of bounds**.
- **Array access with tainted index**.
- **Pointer dereference with tainted offset**.

### Extend Checker

A default Bug Finder analysis might not flag an **Array access out of bounds** issue when the input values are unknown and only a subset of inputs cause the issue. To check for the **Array access out of bounds** issue caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

A default Bug Finder analysis might not flag an **Array access with tainted index** or **Pointer dereference with tainted offset** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Array access out of bounds

#### Issue

**Array access out of bounds** occurs when an array index falls outside the range `[0...array_size-1]` during array access.

#### Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.

- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]); //Noncompliant

}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

#### Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

### Pointer access out of bounds

#### Issue

**Pointer access out of bounds** occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

#### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i; //Noncompliant
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

#### Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

### Array access with tainted index

#### Issue

**Array access with tainted index** detects reading or writing to an array by using a tainted index that has not been validated.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.

- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

#### Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

#### Example - Use Index to Return Buffer Value

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];//Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The function does not check to see if `num` is inside the range of `tab`.

#### Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

### Pointer dereference with tainted offset

#### Issue

**Pointer dereference with tainted offset** detects pointer dereferencing, either reading or writing, using an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

#### Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

#### Example - Dereference Pointer Array

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = new int [SIZE10];
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset];//Noncompliant
        delete pint;
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `i`. The value of `i` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of the index. If the index is inside the valid range, continue with the pointer dereferencing.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

## Check Information

**Group:** 04. Containers (CTR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ARR30-C

# CERT C++: ARR37-C

Do not add or subtract an integer to a pointer to a non-array object

## Description

### Rule Definition

*Do not add or subtract an integer to a pointer to a non-array object.*

### Polyspace Implementation

The rule checker checks for **Invalid assumptions about memory organization**.

## Examples

### Invalid assumptions about memory organization

#### Issue

**Invalid assumptions about memory organization** occurs when you compute the address of a variable in the stack by adding or subtracting from the address of another non-array variable.

#### Risk

When you compute the address of a variable in the stack by adding or subtracting from the address of another variable, you assume a certain memory organization. If your assumption is incorrect, accessing the computed address can be invalid.

#### Fix

Do not perform an access that relies on assumptions about memory organization.

#### Example - Reliance on Memory Organization

```
void func(void) {
    int var1 = 0x00000011, var2;
    *(&var1 + 1) = 0; //Noncompliant
}
```

In this example, the programmer relies on the assumption that `&var1 + 1` provides the address of `var2`. Therefore, an **Invalid assumptions about memory organization** appears on the `+` operation. In addition, a **Pointer access out of bounds** error also appears on the dereference.

#### Correction — Do Not Rely on Memory Organization

One possible correction is not perform direct computation on addresses to access separately declared variables.

## Check Information

**Group:** 04. Containers (CTR)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ARR37-C

# CERT C++: ARR38-C

Guarantee that library functions do not form invalid pointers

## Description

### Rule Definition

*Guarantee that library functions do not form invalid pointers.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Mismatch between data length and size**.
- **Invalid use of standard library memory routine**.
- **Possible misuse of sizeof**.
- **Buffer overflow from incorrect string format specifier**.
- **Invalid use of standard library string routine**.
- **Destination buffer overflow in string manipulation**.
- **Destination buffer underflow in string manipulation**.

## Examples

### Mismatch between data length and size

#### Issue

**Mismatch between data length and size** looks for memory copying functions such as `memcpy`, `memset`, or `memmove`. If you do not control the length argument and data buffer argument properly, Bug Finder raises a defect.

#### Risk

If an attacker can manipulate the data buffer or length argument, the attacker can cause buffer overflow by making the actual data size smaller than the length.

This mismatch in length allows the attacker to copy memory past the data buffer to a new location. If the extra memory contains sensitive information, the attacker can now access that data.

This defect is similar to the SSL Heartbleed bug.

#### Fix

When copying or manipulating memory, compute the length argument directly from the data so that the sizes match.

#### Example - Copy Buffer of Data

```
#include <stdlib.h>
#include <string.h>
```

```
typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    memcpy(&(beta.data[num]), os->data + 2, length); //Noncompliant

    return(1);
}
```

This function copies the buffer `alpha` into a buffer `beta`. However, the `length` variable is not related to `data+2`.

### Correction — Check Buffer Length

One possible correction is to check the length of your buffer against the maximum value minus 2. This check ensures that you have enough space to copy the data to the `beta` structure.

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    if (length<(os->max -2)) {
        memcpy(&(beta.data[num]), os->data + 2, length);
    }

    return(1);

}
```

### Invalid use of standard library memory routine

#### Issue

**Invalid use of standard library memory routine** occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

#### Risk

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Use of Standard Library Memory Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  sscanf("%10c",str1);

  memcpy(str2,str1,6); //Noncompliant

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

#### Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
```

```
{
 /* Fix: Declare str2 with size 6 */
 char str1[12],str2[6];

 printf("Enter string:\n");
 sscanf("%12c",str1);

 memcpy(str2,str1,6);
 return str2;
}
```

### Possible misuse of sizeof

#### Issue

**Possible misuse of sizeof** occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

#### Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

#### Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

#### Example - sizeof Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++) //Noncompliant
    {
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

#### Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

### Buffer overflow from incorrect string format specifier

#### Issue

**Buffer overflow from incorrect string format specifier** occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

#### Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

#### Fix

Use a format specifier that is compatible with the memory buffer size.

#### Example - Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf); //Noncompliant
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

#### Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

### Invalid use of standard library string routine

#### Issue

**Invalid use of standard library string routine** occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source);
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text); //Noncompliant
  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

#### Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Destination buffer overflow in string manipulation

#### Issue

**Destination buffer overflow in string manipulation** occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>
```

```
void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);  //Noncompliant
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

#### Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Destination buffer underflow in string manipulation

#### Issue

**Destination buffer underflow in string manipulation** occurs when certain string manipulation functions write to their destination buffer argument at a negative offset from the beginning of the buffer.

For instance, for the function `sprintf(char* buffer, const char* format)`, you obtain the `buffer` from an operation `buffer = (char*)arr; ... buffer += offset;`. `arr` is an array and `offset` is a negative value.

#### Risk

Buffer underflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer underflow also introduces the risk of code injection.

#### Fix

If the destination buffer argument results from pointer arithmetic, see if you are decrementing a pointer. Fix the pointer decrement by modifying either the original value before decrement or the decrement value.

#### Example - Buffer Underflow in `sprintf` Use

```
#include <stdio.h>
#define offset -2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);  //Noncompliant
}
```

In this example, `&buffer[offset]` is at a negative offset from the memory allocated to `buffer`.

#### Correction — Change Pointer Decrementer

One possible correction is to change the value of `offset`.

```
#include <stdio.h>
#define offset 2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

## Check Information

**Group:** 04. Containers (CTR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ARR38-C

# CERT C++: ARR39-C

Do not add or subtract a scaled integer to a pointer

## Description

### Rule Definition

*Do not add or subtract a scaled integer to a pointer.*

### Polyspace Implementation

The rule checker checks for **Incorrect pointer scaling**.

## Examples

### Incorrect Pointer Scaling

#### Issue

**Incorrect pointer scaling** occurs when you ignore the implicit scaling in pointer arithmetic.

For instance, the defect can occur in these situations.

| Situation | Risk | Possible Fix |
|---|---|---|
| You use the `sizeof` operator in arithmetic operations on a pointer. | The `sizeof` operator returns the size of a data type in number of bytes.<br>When you perform arithmetic operations on a pointer, the argument is implicitly scaled by the size of the data type of the pointed variable. The use of `sizeof` in pointer arithmetic produces unintended results. | Do not use `sizeof` operator in pointer arithmetic. |
| You perform arithmetic operations on a pointer, and then apply a cast. | The implicit scaling in pointer arithmetic depends on the type of an object. Performing these scaled arithmetic and then changing the pointer type by casting might produce unintended results. | Perform the pointer arithmetic after the casting operation. |

#### Fix

The fix depends on the root cause of the defect. Often, the **Result Details** pane shows a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the **Result Details** pane does not show the event history, investigate the root cause of the defect by checking previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example — Use of `sizeof` Operator**

```
#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#include <wchar.h>
#include <stdio.h>



enum { INTBUFSIZE = 80 };
extern int getdata (void);
int buf[INTBUFSIZE];
void foo (void)
{
  int *buf_ptr = buf;

  while (buf_ptr < (buf + sizeof (buf))) {  //Noncompliant
    *buf_ptr++ = getdata ();
  }
}
```

In this example, the operation `sizeof(buf)` is used for obtaining a pointer to the end of the array `buf`. The output of `sizeof(buf)` is scaled by `int`. Because pointer arithmetic is implicitly scaled, the output of `sizeof(buf)` is again scaled by `int` when it is added to `buf`, resulting in unexpected behavior. Polyspace flags the use of `sizeof` operator.

**Correction — Remove `sizeof` Operator**

One possible correction is to use unscaled numbers as offsets.

```
#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#include <wchar.h>
#include <stdio.h>



enum { INTBUFSIZE = 80 };
extern int getdata (void);
int buf[INTBUFSIZE];
void foo (void)
{
  int *buf_ptr = buf;

  while (buf_ptr < (buf + INTBUFSIZE)) {
    *buf_ptr++ = getdata ();
  }
}
```

### Example — Cast Following Pointer Arithmetic

```
int func(void) {
    int x = 0;
    char r = *(char *)(&x + 1); //Noncompliant
    return r;
}
```

In this example, the operation `&x + 1` offsets `&x` by `sizeof(int)`. Following the operation, the resulting pointer points outside the allowed buffer. When you dereference the pointer, a **Pointer access out of bounds** error appears on the `*` operation.

### Correction — Apply Cast Before Pointer Arithmetic

If you want to access the second byte of `x`, first cast `&x` to a `char*` pointer, and then perform the pointer arithmetic. The resulting pointer is offset by `sizeof(char)` bytes and still points within the allowed buffer, whose size is `sizeof(int)` bytes.

```
int func(void) {
    int x = 0;
    char r = *((char *)(&x )+ 1);
    return r;
}
```

### Example — Use of `sizeof` in Function Arguments

```
#include <stddef.h>
#include <stdlib.h>
#include <wchar.h>
enum { WCHAR_BUF = 128 };
FILE* pFile;
//...
void func2_ko (void)
{
    wchar_t error_msg[WCHAR_BUF];
    wcscpy (error_msg, L"Error: ");
    fgetws (error_msg + wcslen (error_msg)   //Noncompliant
           * sizeof (wchar_t), WCHAR_BUF - 7, pFile);
}
```

In this example, an error message is read from the file pointer `pFile` stream and copied to `error_msg` after an offset. The intended offset here is `wcslen(error_msg)`, which is already implicitly scaled when it is added to the `wchar` pointer `error_msg`. Because the offset is then explicitly scaled again by using `sizeof`, Polyspace flags the incorrect scaling.

### Correction — Remove `sizeof` Operator

One possible correction is to remove the `sizeof` operator.

```
#include <stddef.h>
#include <stdlib.h>
#include <wchar.h>
enum { WCHAR_BUF = 128 };
const wchar_t ERROR_PREFIX[8] = L"Error: ";
FILE* pFile;
//...

void func2_ok (void)
{
  const size_t prefix_len = wcslen (ERROR_PREFIX);
  wchar_t error_msg[WCHAR_BUF];
  wcscpy (error_msg, ERROR_PREFIX);
  fgetws (error_msg + prefix_len, WCHAR_BUF - prefix_len, pFile);  //Compliant
  /* ... */
}
```

**Example — Implicitly Scaled Offset When Calling memset**

```
#include <string.h>
#include <stdlib.h>
#include <stddef.h>

#define bigNum unsigned long long

struct Collection {
    bigNum bn_A;
    bigNum bn_B;
    bigNum bn_C;
    int ci_1;
    int ci_2;
};

void foo(void) {
    size_t offset = offsetof(struct Collection, bn_B);
    struct Collection *s = (struct Collection *)malloc(sizeof(struct Collection));
    if (s == NULL) {
        /* Handle malloc() error */
    }

    memset(s + offset, 0, sizeof(struct Collection) - offset); //Noncompliant
    /* ... */
    free(s);
    s = NULL;
}
```

In this example, `offset` is calculated by calling `offsetof`, and then added to `s`. The variable `offset` is the byte offset of `bn_B` in `struct Collection`. When setting memory by using `memset`, instead of offsetting the location by bytes, `offset` is implicitly scaled by the size. The implicit scaling might cause unexpected results. Polyspace raises a violation.

**Correction — Calculate Offset by Using unsigned char***

The violation is caused by the fact that `offset` is scaled by the type of `s`. Avoid the violation by declaring `s` as `unsigned char*`, which is scaled by a factor of one.

```
#include <string.h>
#include <stdlib.h>
#include <stddef.h>

#define bigNum unsigned long long

struct Collection {
    bigNum bn_A;
    bigNum bn_B;
    bigNum bn_C;
    int ci_1;
    int ci_2;
};

void foo(void) {
    size_t offset = offsetof(struct Collection, bn_B);
    unsigned char *s = (unsigned char *)malloc(sizeof(struct Collection));
    if (s == NULL) {
        /* Handle malloc() error */
    }

    memset(s + offset, 0, sizeof(struct Collection) - offset); //Compliant
    /* ... */
    free(s);
    s = NULL;
}
```

## Check Information

**Group:** 04. Containers (CTR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ARR39-C

# CERT C++: CTR50-CPP

Guarantee that container indices and iterators are within the valid range

## Description

### Rule Definition

*Guarantee that container indices and iterators are within the valid range.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access out of bounds**
- **Array access with tainted index**
- **Pointer dereference with tainted offset**
- **Potentially negative container index**

### Extend Checker

A default Bug Finder analysis might not flag an **Array access out of bounds** issue when the input values are unknown and only a subset of inputs cause the issue. To check for the **Array access out of bounds** issue caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

A default Bug Finder analysis might not flag an **Array access with tainted index** or **Pointer dereference with tainted offset** issue for certain inputs that originate outside of the current analysis boundary. See "Sources of Tainting in a Polyspace Analysis". To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Array access out of bounds

#### Issue

**Array access out of bounds** occurs when an array index falls outside the range `[0...array_size-1]` during array access.

#### Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.

- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);  //Noncompliant
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

#### Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
```

```
    int fib[10];

    for (i = 0; i < 10; i++)
     {
         if (i < 2)
             fib[i] = 1;
         else
             fib[i] = fib[i-1] + fib[i-2];
     }

     /* Fix: Print fib[9] instead of fib[10] */
     printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

### Array access with tainted index

#### Issue

**Array access with tainted index** detects reading or writing to an array by using a tainted index that has not been validated.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

#### Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

#### Example - Use Index to Return Buffer Value

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];  //Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The index `num` is obtained from an unsecure source and the function `taintedarrayindex` does not check to see if `num` is inside the range of `tab`.

#### Correction — Check Range Before Use

One possible correction is to check that num is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

### Pointer dereference with tainted offset

#### Issue

**Pointer dereference with tainted offset** detects pointer dereferencing, either reading or writing, using an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

#### Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

#### Example - Dereference Pointer Array

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);
```

```
int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset]; //Noncompliant
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of `offset`. Continue with the pointer dereferencing only if `offset` is inside the valid range.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

### Potentially negative container index

#### Issue

The issue **Potentially negative container index** occurs when:

- You use a potentially negative index to access a standard template library (STL) container that provides random access by using `operator[]`. For example:

```
std::vector v<int>;
int index;
//index becomes negative
v[index]; // index is implicitly cast into unsigned int(index)
```

- You explicitly cast a potentially negative value into an unsigned integer and use the converted value as the index of an STL container. For instance:

```
std::vector v<int>;
int index;
//index  becomes negative
v[static_cast<size_t>(index)]; // size_t has an unsigned type
```

This issue does not apply to associative containers.

**Risk**

STL containers expect an unsigned index. To use a signed integer as an index, the index must be cast to an unsigned integer type, either explicitly or implicitly.

If a negative index is cast to an unsigned index, the resulting unsigned value might be larger than the container expects. Using such a value as the index of an STL container results in an out-of-bounds access. For instance, in the preceding code, if `index` is `-1`, the index of the vector `v` might become `UINT_MAX` after an implicit or explicit cast. The unexpectedly large value of `index` makes the access to `v[index]`out of bounds.

**Fix**

To fix this issue, check the index of the STL container for negative values before accessing the container. Alternatively, use functions that perform range checking, such as the `at()` method of STL containers.

**Example — Potentially Negative Index Causes Out of Bound Access**

In this example, the function `insert_at()` obtains the value of `index` by calling the external function `getIndex()`. The function `insert_at()` checks if the value of `index` is greater than a valid size, but does not check for a negative `index` value. Polyspace reports two violations of this rule in this code:

```
#include <array>
typedef long long LL;
extern LL getIndex();
void insert_at(std::array<int,1024>& container, LL index, int value)
{
    index = getIndex();
    if (index >= container.size())
    {
        // Handle error
        return;
    }

    container[index] = value;  //Noncompliant- a negative index might be
                             // implicitly converted to size_t

    container[static_cast<size_t>(index)] = value; // Noncompliant - a negative
                                                 // index might be converted to size_t
}
```

#### Correction — Use Unsigned Types as Container Index

To fix this issue, declare `index` as an unsigned integer. Explicitly cast the value `getIndex()` returns into `size_t` and check for large values. In this code, if `getIndex()` returns a negative value, `index` becomes unexpectedly large because of the explicit cast. The large value triggers the `if()` condition and the function returns before the defect.

```
#include <array>
typedef long long LL;
typedef unsigned long long ULL;
extern LL getIndex();
void insert_at(std::array<int,1024>& container, ULL index, int value)
{
    index = static_cast<ULL>(getIndex());
    if (index >= container.size())
    {
        // Handle error
        return;
    }

    container[index] = value;  //Compliant
    container[static_cast<size_t>(index)] = value; //Compliant
}
```

#### Correction — Use `at()` Method

The `at()` method of STL containers performs bounds checking. In this code, if `index` becomes unexpectedly large after a cast from signed to unsigned integer, `container.at(index)` raises an `std::out_of_range` exception.

```
#include <array>
typedef long long LL;
extern LL getIndex();
void insert_at(std::array<int,1024>& container, LL index, int value)
{
    index = getIndex();
    if (index >= container.size())
    {
        // Handle error
        return;
    }
    try{
    container.at(index) = value;  //Compliant
    container.at(static_cast<size_t>(index)) = value; //Compliant
    }catch(std::exception& e){
        // check for std::out_of_range
    }
}
```

## Check Information

**Group:** 04. Containers (CTR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CTR50-CPP

# CERT C++: CTR51-CPP

Use valid references, pointers, and iterators to reference elements of a container

## Description

### Rule Definition

*Use valid references, pointers, and iterators to reference elements of a container.*

### Polyspace Implementation

The rule checker checks for **Use of invalid iterator**.

## Examples

### Use of invalid iterator

#### Issue

**Use of invalid iterator** occurs when you use an iterator, pointer, or reference to a container element that has been invalidated by actions such as insertion or erasure. For instance, consider this code:

```
std::vector<int> v = {1,2,3,4,5,6,7};
std::vector<int>::iterator it = std::find(v.begin(), v.end(), 5);
v.push_back(-1);
std::cout<<*it;
```

If the `push_back()` operation causes the vector v to reallocate memory to accommodate a larger size, the iterator `it` is no longer valid. Polyspace flags the dereferencing of invalid iterators.

Certain erasure operations, such as `std::deque::pop_back()` or `std::forward_list::pop_front()` invalidate the iterators to the erased elements. After the erasure, whether any iterators pointed to the erased elements is unknown. If invalidated iterators that point to the erased elements exist in your code, Polyspace does not flag them.

Certain operations return iterators through an `std::pair`. Polyspace does not track iterators that are part of an `std::pair`.

#### Risk

Dereferencing the invalid iterator might produce unexpected results. If the invalidated iterator becomes a dangling pointer or an uninitialized pointer, you might not be able to dereference it safely.

#### Fix

The C++ standard defined which operations invalidate the iterators of the standard library containers. See Containers library. When performing actions that might invalidate an iterator, validate the iterator, for instance, by recalculating its position.

#### Example — Avoid Using Invalid Iterators

```
#include <deque>
```

```
void foo(const int *Value, std::size_t count)
{
  std::deque<int> DQ;
  auto It = DQ.begin();
  for (std::size_t i = 0; i < count; ++i, ++It) {
    DQ.insert(It, Value[i] + 42);//Noncompliant
  }
}
```

In this example, the first `DQ.insert()` operation invalidates the iterator `It`. Its subsequent use might result in undefined behavior. Polyspace flags the use of the invalid operator.

**Correction — Update the Iterator**

When using iterators after an insertion or erasure, update the iterator to avoid undefined behavior. For instance, in this code, `It` is updated after each insertion so that the iterator remains valid.

```
#include <deque>


void foo(const int *Value, std::size_t count)
{
  std::deque<int> DQ;
  auto It = DQ.begin();
  for (std::size_t i = 0; i < count; ++i, ++It) {
    It = DQ.insert(It, Value[i] + 41.0);//Compliant
  }
}
```

## Check Information

**Group:** 04. Containers (CTR)

# Version History

**Introduced in R2022a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CTR51-CPP

# CERT C++: CTR52-CPP

Guarantee that library functions do not overflow

## Description

### Rule Definition

*Guarantee that library functions do not overflow.*[11]

### Polyspace Implementation

The rule checker checks for **Library function overflows sequence container**.

## Examples

### Library Function Overflows Sequence Container

#### Issue

This issue occurs when you use library functions to copy data into a sequence container that might be too small to hold the data. Polyspace reports a violation of this rule when any of these conditions are true:

- The destination container might be too small.
- The size of the destination container might be unknown.

Functions from the C standard library, such as `std::memmove()` or `std::memset()`, overwrite a memory block even if the memory block is invalid. This issue might be prominent when using these functions. Many standard template library (STL) functions are also vulnerable to this issue.

The C++ standard defines vector, list, array, and double ended queue (deque) as sequence containers. See Containers library.

#### Risk

Copying data into a container that is too small results in buffer overflow. Buffer overflow might overwrite unexpected memory locations, resulting in incorrect results, or unexpected program termination.

[11] *This software has been created by MathWorks incorporating portions of: the "SEI CERT-C Website," © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, "SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition," © 2016 Carnegie Mellon University, and "SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition" © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

#### Fix

To prevent container overflow, validate the size of the destination container or the incoming data. For instance, restrict the data being copied to the size of the destination container. Alternatively, allocate sufficient memory for the destination container to safely accommodate the incoming data.

#### Example — Destination Container Too Small

```
#include <algorithm>
#include <vector>
#include <array>

// Destination container size unknown
void copy_vector(const std::vector<int> &src)
{
    std::vector<int> dest;
    std::copy(src.begin(), src.end(), dest.begin()); //Noncompliant
    // ...
}

//Container too small
void fill_vector()
{
    std::vector<int> v;
    std::fill_n(v.begin(), 10, 0x42);    //Noncompliant
}
//Container too small
void copy_array(const std::array<int,10>& src, std::array<int,5>& dest) {

    std::copy(src.begin(), src.end(), dest.begin());     //Noncompliant
}
```

In this example, Polyspace reports violations of this rule on operations that might result in container overflow. For instance:

- In the function `copy_vector`, the size of the destination container `dest` is unknown. Because `std::copy` does not check the bounds of the source container, calling this function to copy `dest` into `src` results in an overflow if `dest` is larger than `src`.
- In the function `fill_vector`, 10 elements are written into the vector `v` before sufficient memory is allocated for the elements. This operation results in an overflow.
- In the function `copy_array`, a container with 10 elements is copied into a container with five elements, resulting in an overflow.

#### Correction — Validate Destination Container Size

Verify that the destination container is large enough so that library functions do not overflow. For instance:

- In `copy_vector`, initialize `dest` with the same size as `src`.
- In `fill_vector`, allocate sufficient memory to `v` to accommodate 10 elements.
- In `copy_array`, use the same size for `src` and `dest`.

```
#include <algorithm>
#include <vector>
#include <array>
```

```
// Destination container size unknown
void copy_vector(const std::vector<int> &src)
{
    std::vector<int> dest(src.size());
    std::copy(src.begin(), src.end(), dest.begin()); //Compliant
    // ...
}

//Container too small
void fill_vector()
{
    std::vector<int> v(10);
    std::fill_n(v.begin(), 10, 0x42);    //Compliant
}
//Container too small

void copy_array(const std::array<int,10>& src, std::array<int,10>& dest) {

    std::copy(src.begin(), src.end(), dest.begin());     //Compliant
}
```

## Check Information

**Group:** Rule 04. Containers (CTR)

# Version History

**Introduced in R2022b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CTR52-CPP

# CERT C++: CTR53-CPP

Use valid iterator ranges

## Description

### Rule Definition

*Use valid iterator ranges.*[12]

### Polyspace Implementation

The rule checker checks for **Invalid iterator usage**.

## Examples

### Invalid iterator usage

#### Issue

This issue occurs when the first and last iterators of a range do not correspond to the `begin` and `end` iterators of the same container. Consider this code:

```
std::vector v{0,1,2,3,4,5,6,7,8,9};
std::vector w{0,1,2,3,4,5,6,7,8,9};
//...
std::for_each(v.end(), v.begin(), [](int i){}); //Noncompliant
std::for_each(v.begin(), w.end(), [](int i){}); //Noncompliant
std::for_each(v.begin(), v.end(), [](int i){}); //Compliant
```

Polyspace reports a violation if the range does not begin and end with the `begin` and `end` iterator of the same container, such as `v.begin()` and `v.end()`. Polyspace checks for the invalid use of most iterators in the standard library. Invalid use of string iterators, such as `std::string` and `std::string_view` iterators are not reported.

#### Risk

- Using an iterator range where the first and last iterators do not refer to the `begin` and `end` iterators of the same container results in undefined behavior. For instance, using `v.begin()` and `w.end()` to delimit a range might result in undefined behavior.

[12] *This software has been created by MathWorks incorporating portions of: the “SEI CERT-C Website,” © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, ”SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition,” © 2016 Carnegie Mellon University, and “SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition” © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

- When you use the end iterator as the first point of a range, the compiler might attempt to access memory that is not part of the container. Accessing memory past the end iterator might result in undefined behavior.

Using such invalid iterator ranges might allow an attacker to run arbitrary code.

**Fix**

To fix violations this rule:

- Use iterators from the same container when defining a range.
- Use the begin iterator as the first element and the end iterator as the last element of the range.

**Example — Invalid Iterator**

```
#include <iostream>
#include <algorithm>
#include <map>

void foo(const std::map<int,int> &a, const std::map<int,int> &b) {
    std::for_each(a.rend(), a.rbegin(), [](std::pair<int,int> i) {  //Noncompliant
    /*...*/});

    std::all_of(a.rbegin(), b.rend(), [](std::pair<int,int> i) {  //Noncompliant
        return 0;});
}
```

In this example, Polyspace reports the invalid use of iterators:

- In the call to the function std::for_each(), the range is delimited by using the a.rend() iterator first and a.rbegin() iterator next. The compiler might increment a.rend() and attempt to access memory that is not part of the container. Polyspace reports a violation.
- In the call to the function std::all_of, the iterators a.rbegin and b.rend() delimits the range. Defining ranges by using two unrelated iterators is undefined behavior. Polyspace reports a violation.

**Correction — Use Iterators From Same Container In Correct Order**

To fix this issue, use iterators from the same container in the correct order when defining a range.

```
#include <iostream>
#include <algorithm>
#include <map>

void foo(const std::map<int,int> &a, const std::map<int,int> &b) {
    std::for_each(a.rbegin(), a.rend(), [](std::pair<int,int> i) {  //Compliant
    /*...*/});

    std::all_of(b.rbegin(), b.rend(), [](std::pair<int,int> i) {  //Compliant
        return 0;});
}
```

## Check Information

**Group:** Rule 04. Containers (CTR)

## Version History

**Introduced in R2022b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CTR53-CPP

# CERT C++: CTR54-CPP

Do not subtract iterators that do not refer to the same container

## Description

### Rule Definition

*Do not subtract iterators that do not refer to the same container.*[13]

### Polyspace Implementation

The rule checker checks for **Subtraction or comparison between iterators from different containers**.

## Examples

### Subtraction or Comparison between Iterators from Different Containers

#### Issue

This issue occurs when you subtract or compare iterators from different containers. For instance:

```
std::vector<int> v1,v2;
//...
std::less<std::vector<int>::iterator>()(v1.begin(), v2.end());
```

Polyspace reports a violation when you compare between iterators from `v1` and `v2`.

#### Risk

If the two iterators are not from the same container, performing subtraction or comparison between these two iterators might result in a nonportable code, undefined behavior, or an unexpected result. For instance:

- Subtracting iterators from different containers results in undefined behavior.
- Comparing iterators from different containers leads to unexpected, implementation-dependent results.

[13] *This software has been created by MathWorks incorporating portions of: the “SEI CERT-C Website,” © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, ”SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition,” © 2016 Carnegie Mellon University, and “SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition” © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

### Fix

Before subtracting or comparing iterators, verify that they belong to the same container.

### Example — Compare or Subtract Pointers to Different Containers

```
#include <iostream>
#include <cassert>
#include <cstddef>
#include <cstdbool>
#include <vector>

template <typename T>
bool isWithinRange(const T *query, const T *begin, size_t span)
{
    return 0 < (query - begin) && ((query - begin) < (std::ptrdiff_t)span); // Noncompliant -unde
}
template <typename T>
bool isOutofRange(const T *query, const T *begin, size_t span)
{
    std::less<const T*> less;
    return less(query, begin) && query > (begin + span); //Noncompliant - unspecified behavior
}


template <typename Iter>
bool inContainerRange(Iter query, Iter begin, Iter end)
{
    return query >= begin && query < end; //Noncompliant - unspecified behavior
}

void foo()
{
    double set[10];
    double *first = &set[0];
    double test_case;
    std::cout << std::boolalpha << isWithinRange(&test_case, first, 10);
    std::cout << std::boolalpha << isOutofRange(&test_case, first, 10);
}
void func(){
    std::vector<double> v1(10);
    std::vector<double> v2(1);
    std::cout << std::boolalpha << inContainerRange(v2.begin(), v1.begin(), v1.end());
}
```

In this example, Polyspace flags comparison and subtraction between iterators or pointers belonging to different containers. For instance:

- In the function `foo()`, `isWithinRange()` is called in such a way that the iterators `query` and `begin` might not point to the same container. The operation `query-begin` might be undefined behavior. Polyspace reports a violation of this rule on the subtraction.
- Similarly, the function `isOutofRange()` is called in such a way that the iterators `query` and `begin` might not point to the same container. The operation `less(query, begin)` might be unspecified behavior. Depending on your hardware architecture and software environment, this comparison might result in an unexpected result or undefined behavior. Polyspace reports a violation of this rule on the comparison.

- In the function `inContainerRange()`, the iterators `query`, `begin`, and `end` might not belong to the same container. The operation `query >= begin` compares iterators from different containers when the function `inContainerRange` is improperly used, as shown in the function `func()`. Comparing iterators from different containers results in an unspecified behavior. Depending on your hardware architecture and software environment, this comparison might result in unexpected result or undefined behavior. Polyspace reports a violation of this rule on the comparison.

#### Correction —Do Not Compare or Subtract Iterators or Pointers to Different Containers

Avoid subtraction or comparison between pointers or iterators that might be from different containers. For instance, range checking is done in the function `isWithinRange()` by using the == operation, which has well-defined behavior even if the operands belong to different containers. While less efficient, this code avoids undefined or unexpected behavior.

```
#include <iostream>
#include <cassert>
#include <cstddef>
#include <cstdbool>
#include <vector>

template <typename T>
bool isWithinRange(const T *query, const T *begin, size_t span)
{
    auto *current = reinterpret_cast<const unsigned char *>(begin);
    auto *end = reinterpret_cast<const unsigned char *>(begin + span);
    auto *ptr = reinterpret_cast<const unsigned char *>(query);
    while(current!=end){
        if(current==ptr){
            return true;
        }
    }

    return false;
}
template <typename T>
bool isOutofRange(const T *query, const T *begin, size_t span)
{
    return !(isWithinRange(query, begin, span));
}


void foo()
{
    double set[10];
    double *first = &set[0];
    double test_case;
    std::cout << std::boolalpha << isWithinRange(&test_case, first, 10);
    std::cout << std::boolalpha << isOutofRange(&test_case, first, 10);
}
```

## Check Information

**Group:** Rule 04. Containers (CTR)

## Version History

**Introduced in R2022b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CTR54-CPP

# CERT C++: CTR55-CPP

Do not use an additive operator on an iterator if the result would overflow

## Description

### Rule Definition

*Do not use an additive operator on an iterator if the result would overflow.*[14]

### Polyspace Implementation

The rule checker checks for **Possible iterator overflow**.

## Examples

### Possible Iterator Overflow

#### Issue

This issue occurs when:

- A constant other than `1`, `0`, or `-1` is added or subtracted from an iterator.
- A variable that is assigned a value other than `1`, `0`, or `-1` is added or subtracted from an iterator. The variable must be local in scope and unmodified before use.

Such operations might result in an iterator before the `begin` iterator or after the `end` iterator of a container. For instance:

```
void print_vec(const std::vector<int> &v) {
    auto end_it = v.begin()+20;//Noncompliant
    for (auto it = v.begin(); it != end_it; ++it) {
        //...
    }
}
```

The function `print_vec` does not check if the iterator `end_it` goes past `v.end()`. The operation `v.begin()+20` might result in an iterator overflow. Polyspace reports a violation on this operation.

[14] *This software has been created by MathWorks incorporating portions of: the "SEI CERT-C Website," © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, "SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition," © 2016 Carnegie Mellon University, and "SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition" © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

Polyspace raises this violation for sequence containers such as `std::vector`, `std::array`, and `std::deque`. As an exception, Polyspace does not report a violation if iterators are incremented or decremented by using the pre or post increment or decrement operators.

#### Risk

The C++ standard does not guarantee the dereferencibility of iterators that are beyond the bounds of the container. An addition or subtraction operation that results in a possible iterator overflow cause an undefined behavior. Often, such an iterator overflow leads to a buffer overflow or buffer underrun. These defects might be exploited by an attacker to execute malicious code.

#### Fix

Validate the addition and subtraction operations on iterators so that the resultant iterators do not overflow.

#### Example — Avoid Addition Operation That Might Overflow on Iterators

```
#include<vector>
#include<string>
#include<iostream>
//Template function that accepts any container and prints the first 20 values
template<typename C, typename T = typename C::value_type>
void checkFirst20(C const& container)
{

  for(auto it = container.begin(), last = it+20; it!=last;++it)//Noncompliant
  {std::cout<<*it<<'\n';}
}

void foo(){
    std::vector<float> v(20);
    checkFirst20(v);
}
```

In this example, the function `checkFirst20()` calculates the value of the iterator `last` adding `20` to another iterator `it`. The code does not verify if `last` remains within the bounds of `container`. Dereferencing such an iterator might result in undefined behavior. Polyspace reports a violation of this rule on this addition.

#### Correction — Validate Addition Operation

To fix this issue, validate the addition operation that defines `last` so that the iterator does not point beyond the bounds of `container`. For instance, in the addition operation that defines `last`, use the smaller of 20 and the value `container.size()` so that `last` is within the bounds of `container`.

```
#include<vector>
#include<string>
#include<iostream>
//Template function that accepts any container and prints the first 20 values
template<typename C, typename T = typename C::value_type>
void printFirst20(C const& container)
{
 typename C::size_type max = 20;
  for(auto it = container.begin(), last = it + std::min(max,container.size()); it!=last;++it)//Co
  {std::cout<<*it<<'\n';}
}
```

```
void foo(){
    std::vector<float> v(5);
    printFirst20(v);
}
```

## Check Information

**Group:** Rule 04. Containers (CTR)

# Version History

**Introduced in R2022b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CTR55-CPP

# CERT C++: CTR56-CPP

Do not use pointer arithmetic on polymorphic objects

## Description

### Rule Definition

*Do not use pointer arithmetic on polymorphic objects.*[15]

### Polyspace Implementation

The rule checker checks for the issue **Pointer Arithmetic on Polymorphic Object**.

## Examples

### Pointer Arithmetic on Polymorphic Object

#### Issue

**Pointer Arithmetic on Polymorphic Object** occurs when you perform pointer arithmetic on a base class pointer pointing to a derived class object. Pointer arithmetic includes array subscripting.

#### Risk

Because the base class and derived class objects might not be the same size, when you attempt pointer arithmetic on a basic class type pointer and this pointer points to a derived class object, the result is undefined behavior.

#### Fix

Typically, the issue occurs when you try to access an array of derived class objects through a base class pointer. To fix this issue, iterate through an array of pointers to objects instead of an array of objects. Instead of a regular array, you can also use a standard template library container such as `std::vector`.

#### Example — Pointer Arithmetic with Polymorphic Objects

In this example, the `Triangle` class is derived from the `Shape` class. The function `example()` takes a `Shape *` pointer as first parameter. A pointer to an array of `Triangle` objects is passed to this

[15] *This software has been created by MathWorks incorporating portions of: the “SEI CERT-C Website,” © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, ”SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition,” © 2016 Carnegie Mellon University, and “SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition” © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

parameter when the function is called from `main()`. The function body contains two instances of pointer arithmetic on the `Shape *` pointer:

- In the initialization statement of the `for` loop, `const Shape* end = exampleShape + count`.
- In the increment statement of the `for` loop, `++exampleShape`.

```
#include <iostream>

class Shape
{
public:
    int height, width;
};

class Triangle : public Shape
{
public:
    double height, width;
};

void example(const Shape* exampleShape, std::size_t count)
{
    for (const Shape* end = exampleShape + count; exampleShape != end; ++exampleShape) //Noncomp
      {
        std::cout << exampleShape->height << std::endl;
    }
}

int main()
{
    Triangle triforce[4] = {};
    example(triforce, 4);
}
```

**Correction — Use an Array of Pointers**

By switching the array of objects in the `for` loop for an array of pointers to objects, you are no longer performing the arithmetic on the objects, but rather on the pointers to the objects. This change avoids the undefined behavior with pointer arithmetic in the `for` loop.

```
#include <iostream>

class Shape
{
public:
    int height, width;
};

class Triangle : public Shape
{
public:
    double height, width;
};

void example(const Shape* const* exampleShape, std::size_t count)
{
    for (const Shape* const* end = exampleShape + count; exampleShape != end; ++exampleShape)
    {
        std::cout << (*exampleShape)->height << std::endl;
    }
}

void main()
{
    Shape* triforce[] = { new Triangle , new Triangle , new Triangle , new Triangle };
    example(triforce, 4);
}
```

**Correction — Use an STL Container**

Instead of an array of objects, the example uses the STL container `std::vector` and fills the vector with pointers to `Triangle` objects. `example()` takes two iterators as parameters, which are the beginning and end of the vector `triforce`. The `for` loop then cycles through the vector of pointers by performing arithmetic on the iterators instead of the polymorphic objects.

```
#include <iostream>
#include <vector>

class Shape
{
public:
    int height, width;
};

class Triangle : public Shape
{
public:
    double height, width;
};

template <typename Iter>

void example(Iter i, Iter e)
{
    for (; i != e; ++i)
    {
        std::cout << (*i)->height << std::endl;
    }
}

void main()
{
    std::vector<Shape*> triforce{ new Triangle , new Triangle , new Triangle , new Triangle };
    example(triforce.cbegin(), triforce.cend());

    for (auto v : triforce)
    {
        delete v;
    }
}
```

## Check Information

**Group:** Rule 04. Containers (CTR)

## Version History

**Introduced in R2023a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CTR56-CPP

# CERT C++: CTR57-CPP

Provide a valid ordering predicate

## Description

### Rule Definition

*Provide a valid ordering predicate.*[16]

### Polyspace Implementation

The rule checker checks for **Use of predicate lacking strict weak ordering**

## Examples

### Use of predicate lacking strict weak ordering

#### Issue

**Use of predicate lacking strict weak ordering** occurs when you use one of these compare types as predicates in standard template library (STL) algorithms and containers:

- `std::less_equal`
- `std::greater_equal`
- `std::equal_to`
- `std::not_equal_to`
- `std::logical_or`
- `std::logical_and`

For a list of standard library algorithms that expect a predicate with strict weak ordering, see Compare.

If you use a user-defined predicate function, Polyspace does not check if the custom predicate adheres to strict weak ordering.

[16] *This software has been created by MathWorks incorporating portions of: the “SEI CERT-C Website,” © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, ”SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition,” © 2016 Carnegie Mellon University, and “SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition” © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

#### Risk

Algorithms and containers of the standard template library use predicates to sort and compare their elements. The predicates must adhere to strict weak ordering. That is, the predicate must adhere to these requirements:

- Irreflexivity: For all x, comparing x to itself must always evaluate to false.
- Assymetry: Far all x, y: if comparing x to y evaluates to true, then comparing y to x must evaluate to false.
- Transitivity: For all x, y, z: if comparing x to y and y to z both evaluate to true, then comparing x to z must also evaluate to true.

These compare type methods violate at least one of these requirements:

- std::less_equal
- std::greater_equal
- std::equal_to
- std::not_equal_to
- std::logical_or
- std::logical_and

Using the preceding predicates with algorithms and containers from the standard library might result in infinite loops, erratic behavior, and bugs that are difficult to diagnose.

#### Fix

To resolve this defect, avoid using the preceding methods as predicates for algorithms and containers from the standard template library. Instead, use methods that adhere to strict weak ordering. For instance, use functions such as std::less or std::greater. The STL uses these functions or their equivalent operators as the default ordering predicate. Using the default predicate might be sufficient to resolve the defect.

#### Example — Avoid Using Predicates That Do Not Adhere to Strict Weak Ordering with STL Algorithms

```
#include <functional>
#include <iostream>
#include <set>
#include <map>

int main(void)
{
    // GE
    std::set<int, std::greater_equal<int>>//Noncompliant
        s1{2, 5, 8};
    auto r = s1.equal_range(5);
    //returns 0
    std::cout << std::distance(r.first, r.second) << std::endl;

    //LE
    std::map<int, std::string, std::less_equal<int>>//Noncompliant
        m1{{2, "AB"}, {5, "CD"}, {8, "EF"}};
    auto r3 = m1.equal_range(5);
    //returns 0
    std::cout << std::distance(r3.first, r3.second) << std::endl;
    return 0;
}
```

In this example, Polyspace flags the use of inappropriate predicates when declaring STL containers. For instance:

- The predicate `greater_equal` does not return false when comparing the same objects, violating the irreflexivity requirement.
- The predicate `less_equal` does not return false when comparing the same objects, violating the irreflexivity requirement.

Polyspace flags the use of these function as predicates when declaring containers such as `std::set` or `std::map`.

#### Correction — Use the Default Predicate

By default, these containers use the function `std::less` as the ordering predicate. Because this function adheres to strict weak ordering relation, Polyspace does not raise a violation when you declare the containers by using the default predicate.

```
#include <functional>
#include <iostream>
#include <set>
#include <map>

int main(void)
{
    std::set<int> s2{2, 5, 8}; //Compliant
    auto r2 = s2.equal_range(5);
    //returns 1
    std::cout << std::distance(r2.first, r2.second) << std::endl;


    std::map<int, std::string> m2{{2, "AB"}, {5, "CD"}, {8, "EF"}}; //Compliant
    auto r4 = m2.equal_range(5);
    //returns 1
    std::cout << std::distance(r4.first, r4.second) << std::endl;

    return 0;
}
```

## Check Information

**Group:** Rule 04. Containers (CTR)

# Version History

**Introduced in R2022a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

CTR57-CPP

# CERT C++: CTR58-CPP

Predicate function objects should not be mutable

## Description

### Rule Definition

*Predicate function objects should not be mutable.*[17]

### Polyspace Implementation

The rule checker checks for **Function object that modifies its state**.

## Examples

### Function object that modifies its state

#### Issue

This issue occurs when the following conditions are true:

- You use a STL algorithm that takes a predicate function object. For a full list of STL algorithms, see Algorithms library.
- The state of the function object is modified, where the state is one of the following:
  - The `this` pointer if the predicate object is an instance of a class that defines a function call operator `operator()`.
  - A captured-by-copy value if the predicate object is a lambda function. A lambda function that captures values by reference is compliant.

A function object state is modified if it is written to or if a non-const method is called on it. For example, in this code snippet, non-const method `increment()` modifies the state `elemPosition` of `operator()` :

```
#include <functional>
#include <vector>
#include <algorithm>
```

[17] *This software has been created by MathWorks incorporating portions of: the “SEI CERT-C Website,” © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, ”SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition,” © 2016 Carnegie Mellon University, and “SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition” © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

```
class myPred : public std::unary_function<int, bool> {
  public:
    myPred(): elemPosition(0) {}
    void increment() {++elemPosition;} // Non-const method
    bool operator()(const int&) // function call operator
    {
        increment(); // method called on state elemPosition
        return elemPosition == 3;
    }
  private:
    size_t elemPosition;
};

void func(std::vector<int> v) {
    std::remove_if(v.begin(), v.end(), myPred()); // Noncompliant
}
```

Polyspace flags function objects that modify their state even if the function object is not a predicate.

Polyspace does not flag function objects that are wrapped in these:

- `std::ref`
- `std::cref`
- `std::function`
- `std::bind`
- `std::not1`

**Risk**

Predicate function objects take a single argument and use that argument to return a value that is testable as a boolean (true or false). Standard Template Library (STL) algorithms that accept a predicate function object may, depending on the algorithm implementation, make a copy of the function object. The invocation of the copy might cause unexpected results.

For example, suppose that you use an algorithm to remove one element from a list. To determine the element to remove, the algorithm uses a predicate function object that modifies a state related to the function object identity each time the function object is called. You might expect the predicate object to return true only once. If the algorithm makes a copy of the predicate object, the state of the copy is also modified and the copy returns true a second time, removing a second element from the list.

**Fix**

To avoid unexpected results, consider one of the following:

- Wrap the predicate function object in a `std::reference_wrapper<T>` before you pass it to the algorithm. If the wrapper object is copied, all copies refer to the same underlying predicate function object.

  For example, in this code snippet, predicate function object `myObj` is wrapped in `std::ref` when passed to `std::remove_if`.

  ```
  class predObj {
      // Defines function object that modifies its state
  };
  void func() {
      std::vector<int> v{0, 1, 2, 3, 4, 5};
      //
      predObj myObj;
  ```

```
    v.erase(std::remove_if(v.begin(), v.end(), std::ref(myObj)), v.end());
    //....
}
```

- Implement a `const` function call operator that does not modify the state of the predicate object. For example, in this code snippet, `predModifies` defines a call operator that modifies `elemPosition` before checking the equality whereas `predDoesNotModify` checks only the equality of `var` without modifying it.

  ```
  #include <functional>
  #include <vector>
  #include <algorithm>

  class predModifies : public std::unary_function<int, bool>
  {
  public:
  predModifies() : elemPosition(0) {}
  bool operator()(const int &) const { return (++elemPosition) == 3; }
  //call operator modifies elemPosition
  private:
  mutable size_t elemPosition;
  };

  class predDoesNotModify: public std::unary_function<int, bool>
  {
  public:
  bool operator()(const int& var) const { return var == 3; }
  //call operator does not modify state of object
  };
  ```

**Example — Function Object that Modifies Its State**

```
#include <functional>
#include <vector>
#include <algorithm>


void lambdaDoesNotModify(std::vector<int> v)
{
  int elemPosition = 0;
  std::remove_if(v.begin(), v.end(),
    [elemPosition](const int & element) mutable {  //Compliant
      return elemPosition == 0 && element == 3;
    });
}

void lambdaCaptureByRef(std::vector<int> v)
{
  int elemPosition = 0;
  std::remove_if(v.begin(), v.end(),
    [&elemPosition](const int & i) { // Compliant
      return ++elemPosition == 3;
    });
}

class nonConstPred : public std::unary_function<int, bool>
{
  public:
    nonConstPred() : elemPosition(0) {}
    void increment() { ++elemPosition; }
```

```
    bool operator()(const int &)
    {
      increment();
      return elemPosition == 3;
    }
  private:
    size_t elemPosition;
};

void myFunc(std::vector<int> v)
{
  std::remove_if(v.begin(), v.end(), nonConstPred()); // Noncompliant
}
```

In this example, the use of predicate function object `nonConstPred()` inside function `myFunc` is not compliant. The call operator inside class `nonConstPred` uses a non-const method `increment()` that modifies `elemPosition`, the state of the predicate. If the algorithm `std::remove_if` makes a copy of the predicate function object, there might be two instances of `elemPosition == 3` being true, which might cause unexpected results.

The use of a lambda function as predicate function object in function `lambdaDoesNotModify` is compliant because the state of the function object is not modified. Similarly, the lambda function in `lambdaCaptureByRef` is compliant because the state of the function object is captured by reference.

### Correction — Wrap the Function Object in `std::ref`

```
#include <functional>
#include <vector>
#include <algorithm>


void lambdaDoesNotModify(std::vector<int> v)
{
  int elemPosition = 0;
  std::remove_if(v.begin(), v.end(),
    [elemPosition](const int & element) mutable {  //Compliant
      return elemPosition == 0 && element == 3;
    });
}

void lambdaCaptureByRef(std::vector<int> v)
{
  int elemPosition = 0;
  std::remove_if(v.begin(), v.end(),
    [&elemPosition](const int & i) { // Compliant
      return ++elemPosition == 3;
    });
}

class nonConstPred : public std::unary_function<int, bool>
{
  public:
    nonConstPred() : elemPosition(0) {}
    void increment() { ++elemPosition; }
    bool operator()(const int &)
    {
      increment();
      return elemPosition == 3;
    }
  private:
    size_t elemPosition;
};

void myFunc(std::vector<int> v)
{
  nonConstPred myPred;
  std::remove_if(v.begin(), v.end(), std::ref(myPred)); // Compliant
}
```

One possible correction is to wrap the function object in `std::ref`. All copies of the wrapper object refer to the same underlying function object and there is only one instance of the state `elemPosition`.

### Check Information

**Group:** Rule 04. Containers (CTR)

## Version History

**Introduced in R2022a**

### See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

CTR58-CPP

# CERT C++: STR30-C

Do not attempt to modify string literals

## Description

### Rule Definition

*Do not attempt to modify string literals.*

### Polyspace Implementation

The rule checker checks for **Writing to const qualified object**.

## Examples

### Writing to const qualified object

#### Issue

**Writing to `const` qualified object** occurs when you do one of the following:

- Use a `const`-qualified object as the destination of an assignment.
- Pass a `const`-qualified object to a function that modifies the argument.

For instance, the defect can occur in the following situations:

- You pass a `const`-qualified object as first argument of one of the following functions:
  - `mkstemp`
  - `mkostemp`
  - `mkostemps`
  - `mkdtemp`
- You pass a `const`-qualified object as the destination argument of one of the following functions:
  - `strcpy`
  - `strncpy`
  - `strcat`
  - `memset`
- You perform a write operation on a `const`-qualified object.

#### Risk

The risk depends upon the modifications made to the `const`-qualified object.

| Situation | Risk |
|---|---|
| Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on. | These functions replace the last six characters of their first argument with a string. Therefore, they expect a modifiable `char` array as their first argument. |
| Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on. | These functions modify their destination argument. Therefore, they expect a modifiable `char` array as their destination argument. |
| Writing to the object | The `const` qualifier implies an agreement that the value of the object will not be modified. By writing to a `const`-qualified object, you break the agreement. The result of the operation is undefined. |

#### Fix

The fix depends on the modification made to the `const`-qualified object.

| Situation | Fix |
|---|---|
| Passing to `mkstemp`, `mkostemp`, `mkostemps`, `mkdtemp`, and so on. | Pass a non-`const` object as first argument of the function. |
| Passing to `strcpy`, `strncpy`, `strcat`, `memset` and so on. | Pass a non-`const` object as destination argument of the function. |
| Writing to the object | Perform the write operation on a non-`const` object. |

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Writing to const-Qualified Object

```
#include <string.h>

const char* buffer = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string); //Noncompliant
}
```

In this example, because `buffer` is `const`-qualified, `strchr(buffer,'X')` returns a `const`-qualified `char*` pointer. When this `char*` pointer is used as the destination argument of `strcpy`, a **Writing to const qualified object** error appears.

### Correction — Copy `const`-Qualified Object to Non-`const` Object

One possible correction is to assign the constant string to a non-`const` object and use the non-`const` object as destination argument of `strchr`.

```
#include <string.h>

char buffer[] = "abcdeXXXXXXX";

void func(char* string) {
    char *ptr = (char*)strchr(buffer,'X');
    if(ptr)
        strcpy(ptr,string);
}
```

## Check Information

**Group:** 05. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

STR30-C

# CERT C++: STR31-C

Guarantee that storage for strings has sufficient space for character data and the null terminator

## Description

### Rule Definition

*Guarantee that storage for strings has sufficient space for character data and the null terminator.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Use of dangerous standard function**.
- **Buffer overflow from incorrect string format specifier**.
- **Destination buffer overflow in string manipulation**.
- **Insufficient destination buffer size**.

## Examples

### Use of dangerous standard function

#### Issue

The **Use of dangerous standard function** check highlights uses of functions that are inherently dangerous or potentially dangerous given certain circumstances. The following table lists possibly dangerous functions, the risks of using each function, and what function to use instead.

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `gets` | Inherently dangerous — You cannot control the length of input from the console. | `fgets` |
| `std::cin::operator>>` and `std::wcin::operator>>` | Inherently dangerous — You cannot control the length of input from the console. | Preface calls to `cin` by `cin.width` to control the input length. This method can result in truncated input.<br>To avoid potential buffer overflow and truncated input, use `std::string` objects as destinations for >> operator. |
| `strcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `strncpy` |
| `stpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `stpncpy` |

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `lstrcpy` or `StrCpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy` |
| `strcat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `strncat`, `strlcat`, or `strcat_s` |
| `lstrcat` or `StrCat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat` |
| `wcpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcpncpy` |
| `wcscat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `wcsncat`, `wcslcat`, or `wcncat_s` |
| `wcscpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcsncpy` |
| `sprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `snprintf` |
| `vsprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `vsnprintf` |

**Risk**

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

**Fix**

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Using `sprintf`**

```
#include <stdio.h>
#include <string.h>
```

```
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1) //Noncompliant
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

#### Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

### Buffer overflow from incorrect string format specifier

#### Issue

**Buffer overflow from incorrect string format specifier** occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

#### Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

#### Fix

Use a format specifier that is compatible with the memory buffer size.

#### Example - Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf); //Noncompliant
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

#### Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

### Destination buffer overflow in string manipulation

#### Issue

**Destination buffer overflow in string manipulation** occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string); //Noncompliant
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

#### Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Insufficient destination buffer size

#### Issue

**Insufficient destination buffer size** occurs when the destination buffer in a `strcpy` operation cannot accommodate the source buffer and a null terminator. This issue is reported if the size of the source buffer is unknown. Consider this code:

```
int main (int argc, char *argv[])
{
  const char *const input = ((argc && argv[0]) ? argv[0] : "");
  char str[100];
  strcpy(input, str); // Noncompliant
}
```

In this case, the size of the source buffer `input` is unknown. The size of the destination buffer `str` might be smaller than the value `(strlen(input)+1)`. Polyspace reports a violation on the `strcpy` operation.

#### Risk

Using a destination buffer of insufficient size might allow an attacker to cause a buffer overflow. In the preceding code example, if `argv[0]` contains 100 or more characters, the `strcpy` operation results in a buffer overflow.

#### Fix

Before calling the function `strcpy()`, allocate sufficient memory dynamically. For instance, use the function `strlen()` to determine the size of the source buffer and then allocate the destination buffer so that its size is greater than the value `strlen(source) + 1`.

#### Example — Destination Buffer Too Small

In this example, the size of the `source` buffer is unknown, while the size of the `destination` buffer is fixed at `128`. The size of the `destination` buffer might not be sufficient to accommodate the characters from the `source` buffer and terminate the buffer with a null. Polyspace reports a violation of the rule.

```
#include <cstring>

int main(int argc, char *argv[]) {
    const char *const source = (argc && argv[0]) ? argv[0] : "";
    char destination[128];
    strcpy(const_cast<char*>(source), destination);//Noncompliant
    return 0;
}
```

#### Correction — Allocate Sufficient Memory for Destination Buffer

This violation is resolved by allocating sufficient memory for the `destination` buffer. For instance, use the function `strlen()` to calculate the size of the `source` buffer and allocate sufficient memory for the `destination` buffer so that it can accommodate all characters from the `source` buffer and the null terminator (`'\0'` ).

```
#include <cstring>
#include <cstdlib>

int main(int argc, char *argv[]) {
    const char *const source = (argc && argv[0]) ? argv[0] : "";
    char* destination = (char *)malloc(strlen(source)+ 1);
    if(destination!=NULL){
        strcpy(const_cast<char*>(source), destination);//Compliant
    }else{
        /*Handle Error*/
    }
    //...
    free(destination);
    return 0;
}
```

## Check Information

**Group:** 05. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

STR31-C

# CERT C++: STR32-C

Do not pass a non-null-terminated character sequence to a library function that expects a string

## Description

### Rule Definition

*Do not pass a non-null-terminated character sequence to a library function that expects a string.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Invalid use of standard library string routine**.
- **Tainted NULL or non-null-terminated string**.

### Extend Checker

A default Bug Finder analysis might not flag a **Tainted NULL or non-null-terminated string** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Invalid use of standard library string routine

#### Issue

**Invalid use of standard library string routine** occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source);
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);  //Noncompliant
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

#### Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Tainted NULL or non-null-terminated string

#### Issue

**Tainted NULL or non-null-terminated string** looks for strings from unsecure sources that are being used in string manipulation routines that implicitly dereference the string buffer. For example, `strcpy` or `sprintf`.

**Tainted NULL or non-null-terminated string** raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

**Note** If you reference a string using the form `ptr[i]`, `*ptr`, or pointer arithmetic, Bug Finder raises a **Use of tainted pointer** defect instead. The **Tainted NULL or non-null-terminated string** defect is raised only when the pointer is used as a string.

#### Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

#### Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

#### Example - Getting String from Input Argument

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
      char userstr[MAX];
      read(0,userstr,MAX);
      char str[SIZE128] = "Warning: ";
      strncat(str, userstr, SIZE128-(strlen(str)+1)); //Noncompliant
      print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

#### Correction — Validate the Data

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // Noncompliant
    //TAINTED_STRING confined to the sanitizer function.
        // - string is not null
        // - string has a positive and limited size
        // - TAINTED_STRING on strlen used as a firewall
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

#### Correction — Validate the Data

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

## Check Information

**Group:** 05. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

STR32-C

# CERT C++: STR34-C

Cast characters to unsigned char before converting to larger integer sizes

## Description

### Rule Definition

*Cast characters to unsigned char before converting to larger integer sizes.*

### Polyspace Implementation

The rule checker checks for **Misuse of sign-extended character value**.

## Examples

### Misuse of sign-extended character value

#### Issue

**Misuse of sign-extended character value** occurs when you convert a signed or plain `char` data type to a wider integer data type with sign extension. You then use the resulting sign-extended value as array index, for comparison with EOF or as argument to a character-handling function.

#### Risk

*Comparison with EOF*: Suppose, your compiler implements the plain `char` type as signed. In this implementation, the character with the decimal form of 255 (–1 in two’s complement form) is stored as a signed value. When you convert a `char` variable to the wider data type `int` for instance, the sign bit is preserved (sign extension). This sign extension results in the character with the decimal form 255 being converted to the integer –1, which cannot be distinguished from EOF.

*Use as array index*: By similar reasoning, you cannot use sign-extended plain `char` variables as array index. If the sign bit is preserved, the conversion from `char` to `int` can result in negative integers. You must use positive integer values for array index.

*Argument to character-handling function*: By similar reasoning, you cannot use sign-extended plain `char` variables as arguments to character-handling functions declared in `ctype.h`, for instance, `isalpha()` or `isdigit()`. According to the C11 standard (Section 7.4), if you supply an integer argument that cannot be represented as `unsigned char` or EOF, the resulting behavior is undefined.

#### Fix

Before conversion to a wider integer data type, cast the signed or plain `char` value explicitly to `unsigned char`.

#### Example - Sign-Extended Character Value Compared with EOF

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];
```

```
static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = *buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {  //Noncompliant
        /* Handle error */
        fatal_error();
    }
}
```

In this example, the function `parser` can traverse a string input `buf`. If a character in the string has the decimal form 255, when converted to the `int` variable `c`, its value becomes –1, which is indistinguishable from `EOF`. The later comparison with `EOF` can lead to a false positive.

#### Correction — Cast to `unsigned char` Before Conversion

One possible correction is to cast the plain `char` value to `unsigned char` before conversion to the wider `int` type.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = (unsigned char)*buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

## Check Information

**Group:** 05. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

STR34-C

# CERT C++: STR37-C

Arguments to character-handling functions must be representable as an unsigned char

## Description

### Rule Definition

*Arguments to character-handling functions must be representable as an unsigned char.*

### Polyspace Implementation

The rule checker checks for **Invalid arguments to character-handling functions**.

## Examples

### Invalid arguments to character-handling functions

#### Issue

**Invalid arguments to character-handling functions** occurs when you use a signed or plain `char` variable with a negative value as argument to a character-handling function declared in `<cctype>`, for instance, `isalpha()` or `isdigit()`.

#### Risk

You cannot use plain `char` variables as arguments to these character-handling functions. On certain platforms, plain `char` variables can have negative values that cannot be represented as `unsigned char` or EOF, resulting in undefined behavior.

#### Fix

To avoid unexpected results, explicitly cast plain `char` variables to unsigned `char` before passing to character-handling functions.

## Check Information

**Group:** 05. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

STR37-C

# CERT C++: STR38-C

Do not confuse narrow and wide character strings and functions

## Description

### Rule Definition

*Do not confuse narrow and wide character strings and functions.*

### Polyspace Implementation

The rule checker checks for **Misuse of narrow or wide character string**.

## Examples

### Misuse of narrow or wide character string

#### Issue

**Misuse of narrow or wide character string** occurs when you pass a narrow character string to a wide string function, or a wide character string to a narrow string function.

**Misuse of narrow or wide character string** raises no defect on operating systems where narrow and wide character strings have the same size.

#### Risk

Using a narrow character string with a wide string function, or vice versa, can result in unexpected or undefined behavior.

If you pass a wide character string to a narrow string function, you can encounter these issues:

- Data truncation. If the string contains null bytes, a copy operation using `strncpy()` can terminate early.
- Incorrect string length. `strlen()` returns the number of characters of a string up to the first null byte. A wide string can have additional characters after its first null byte.

If you pass a narrow character string to a wide string function, you can encounter this issue:

- Buffer overflow. In a copy operation using `wcsncpy()`, the destination string might have insufficient memory to store the result of the copy.

#### Fix

Use the narrow string functions with narrow character strings. Use the wide string functions with wide character strings.

#### Example - Passing Wide Character Strings to `strncpy()`

```
#include <string.h>
#include <wchar.h>

void func(void)
```

```
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    strncpy(reinterpret_cast<char *>(wide_str2), reinterpret_cast<const char *>(wide_str1), 10);
}
```

In this example, `strncpy()` copies 10 wide characters from `wide_strt1` to `wide_str2`. If `wide_str1` contains null bytes, the copy operation can end prematurely and truncate the wide character string.

#### Correction — Use `wcsncpy()` to Copy Wide Character Strings

One possible correction is to use `wcsncpy()` to copy `wide_str1` to `wide_str2`.

```
#include <string.h>
#include <wchar.h>

void func(void)
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    wcsncpy(wide_str2, wide_str1, 10);
}
```

## Check Information

**Group:** 05. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

STR38-C

# CERT C++: STR50-CPP

Guarantee that storage for strings has sufficient space for character data and the null terminator

## Description

### Rule Definition

*Guarantee that storage for strings has sufficient space for character data and the null terminator.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Use of dangerous standard function**.
- **Buffer overflow from incorrect string format specifier**.
- **Destination buffer overflow in string manipulation**.
- **Insufficient destination buffer size**.
- **Input string not null-terminated**

## Examples

### Use of dangerous standard function

#### Issue

The **Use of dangerous standard function** check highlights uses of functions that are inherently dangerous or potentially dangerous given certain circumstances. The following table lists possibly dangerous functions, the risks of using each function, and what function to use instead.

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `gets` | Inherently dangerous — You cannot control the length of input from the console. | `fgets` |
| `std::cin::operator>>` and `std::wcin::operator>>` | Inherently dangerous — You cannot control the length of input from the console. | Preface calls to `cin` by `cin.width` to control the input length. This method can result in truncated input.<br>To avoid potential buffer overflow and truncated input, use `std::string` objects as destinations for >> operator. |
| `strcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `strncpy` |

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `stpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `stpncpy` |
| `lstrcpy` or `StrCpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy` |
| `strcat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `strncat`, `strlcat`, or `strcat_s` |
| `lstrcat` or `StrCat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat` |
| `wcpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcpncpy` |
| `wcscat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `wcsncat`, `wcslcat`, or `wcncat_s` |
| `wcscpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcsncpy` |
| `sprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `snprintf` |
| `vsprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `vsnprintf` |

#### Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.

- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Example - Using `sprintf`

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1) //Noncompliant
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

### Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

### Buffer overflow from incorrect string format specifier

#### Issue

**Buffer overflow from incorrect string format specifier** occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

#### Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

#### Fix

Use a format specifier that is compatible with the memory buffer size.

#### Example - Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf); //Noncompliant
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

#### Correction — Use Smaller Precision in Format Specifier

One possible correction is to use a smaller precision in the format specifier.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

### Destination buffer overflow in string manipulation

#### Issue

**Destination buffer overflow in string manipulation** occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example - Buffer Overflow in `sprintf` Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string); //Noncompliant
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

#### Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Insufficient destination buffer size

#### Issue

**Insufficient destination buffer size** occurs when the destination buffer in a `strcpy` operation cannot accommodate the source buffer and a null terminator. This issue is reported if the size of the source buffer is unknown. Consider this code:

```
int main (int argc, char *argv[])
{
  const char *const input = ((argc && argv[0]) ? argv[0] : "");
  char str[100];
  strcpy(input, str); // Noncompliant
}
```

In this case, the size of the source buffer `input` is unknown. The size of the destination buffer `str` might be smaller than the value `(strlen(input)+1)`. Polyspace reports a violation on the `strcpy` operation.

#### Risk

Using a destination buffer of insufficient size might allow an attacker to cause a buffer overflow. In the preceding code example, if argv[0] contains 100 or more characters, the strcpy operation results in a buffer overflow.

#### Fix

Before calling the function strcpy(), allocate sufficient memory dynamically. For instance, use the function strlen() to determine the size of the source buffer and then allocate the destination buffer so that its size is greater than the value strlen(source) + 1.

#### Example — Destination Buffer Too Small

In this example, the size of the source buffer is unknown, while the size of the destination buffer is fixed at 128. The size of the destination buffer might not be sufficient to accommodate the characters from the source buffer and terminate the buffer with a null. Polyspace reports a violation of the rule.

```
#include <cstring>

int main(int argc, char *argv[]) {
    const char *const source = (argc && argv[0]) ? argv[0] : "";
    char destination[128];
    strcpy(const_cast<char*>(source), destination);//Noncompliant
    return 0;
}
```

#### Correction — Allocate Sufficient Memory for Destination Buffer

This violation is resolved by allocating sufficient memory for the destination buffer. For instance, use the function strlen() to calculate the size of the source buffer and allocate sufficient memory for the destination buffer so that it can accommodate all characters from the source buffer and the null terminator ('\0' ).

```
#include <cstring>
#include <cstdlib>

int main(int argc, char *argv[]) {
    const char *const source = (argc && argv[0]) ? argv[0] : "";
    char* destination = (char *)malloc(strlen(source)+ 1);
    if(destination!=NULL){
        strcpy(const_cast<char*>(source), destination);//Compliant
    }else{
        /*Handle Error*/
    }
    //...
    free(destination);
    return 0;
}
```

### Input string not null-terminated

#### Issue

The issue **Input string not null-terminated** occurs when both of these conditions are true:

- You obtain a string from an input function that does not necessarily terminates the string by a null character.

- You use the input string that does not have the terminating null character in places where a null-terminated string is expected.

For instance:

```
char str[10];
scanf("%10c", str);
std::string S(str);//Null-terminated string expected
```

Here, the constructor of `std::string` expects a string terminated by a null character, but `str` is not null-terminated. Polyspace reports a violation of this rule.

#### Risk

Functions in the C++ library assume that a valid string is terminated by a null character and apply this assumption string algorithms. Using these functions and algorithms with a string that lacks a terminating null character results in undefined behavior.

#### Fix

The fix for this issue depends on the context:

- You can fix this issue by terminating raw input strings by a null. For example, this code declares `str` with 11 elements. After filling the first 10 elements with the raw input, assign the last element the value `'\0'`. After manually terminating the input string with a null character, you can use the input string with standard template library (STL) string functions without violating this rule.

  ```
  char str[11];
  scanf("%10c", str);
  str[10] = '\0';
  std::string S(str);//Valid string
  ```
- Some C++ functions accepts character pointers or substrings as input. These functions can correctly handle a raw character array without a terminating null character. For instance, This code constructs the `std::string` object `S` by using a nondefault constructor. This constructor allows the use of `str`, which lacks a terminating null character. Such uses do not violate this rule.

  ```
  char str[10];
  scanf("%10c", str);
  std::string S(str,sizeof(str));//Valid use of substring
  ```

#### Example — Print Input String Without Terminating Null Character

In this example, the `read` function accepts an array of characters, which is then stored in the `char` array `buffer`. Because `buffer`does not have a terminating null character, using `buffer` with the function `printf()` results in undefined behavior. Polyspace reports a violation of this rule.

```
#include <iostream>
#include <string>
#include <fstream>
#include <cstdio>
void echo_instream(std::istream& instream) {
    //...
    char buffer[10];
    instream.read(buffer,sizeof(buffer));
    //...
    printf("%s", buffer); //Noncompliant
    //...
```

```
}
```

#### Correction — Terminate Input String with `'\0'`

To fix this issue, manually add the terminating null character at the end of `buffer`.

```
#include <iostream>
#include <string>
#include <fstream>
#include <cstdio>
void echo_instream(std::istream& instream) {
    //...
    char buffer[10];
    instream.read(buffer,sizeof(buffer)-1);
    buffer[9] = '\0';
    //...
    printf("%s", buffer); //Compliant
    //...
}
```

## Check Information

**Group:** 05. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

STR50-CPP

# CERT C++: STR51-CPP

Do not attempt to create a `std::string` from a null pointer

## Description

### Rule Definition

*Do not attempt to create a `std::string` from a null pointer.*

### Polyspace Implementation

The rule checker checks for the issue **String operations on null pointer**.

## Examples

### String Operations On Null Pointer

#### Issue

This issue occurs when:

- You perform string operations that require calling `std::char_traits::length()` on `NULL`, 0, or `nullptr` . Examples of such string operations are creating, appending, assigning, inserting, or replacing the string. For a list of operations that results in call to `std::char_traits::length()`, see STR51-CPP.

  This issue is a specific instance of the issue **Null pointer**, which causes violations of `CERT C++: EXP34-C`. Consider this code:

  ```
  std::string getString(); //returns nullptr
  void foo(){
      std::string str{getString()};//Defect
  }
  ```

  Construction of `str` requires an implicit call to `std::char_traits::length()`. Polyspace reports a violation because `getString()` returns a `nullptr`, which results in calling `std::char_traits::length()` on `nullptr`.
- You perform certain string operations on a nonnull pointer to an uninitialized memory block. Consider this code:

  ```
  void foo() {

      const char* uninitialized = (const char*)std::malloc(size*sizeof(char) + 1);;
       std::string tmp(uninitialized);  //Noncompliant
  }
  ```

  Polyspace reports a violation of this rule when `tmp` is constructed by using the uninitialized memory in `uninitialized`.

A violation of this rule is not reported for stubbed functions.

#### Risk

Performing string operations that require calling `std::char_traits::length()` on `NULL`, 0, or `nullptr` might result in an undefined behavior. The function `std::char_traits::length()` dereferences the null pointer, which is an undefined behavior.

Performing string operations on uninitialized memory results in unexpected outcome and might result in bugs that are difficult to diagnose.

#### Fix

Check if the string object is a null pointer or an empty string before you perform string operations.

#### Example — String Operations Using Null Pointer

```
#include <cstdlib>
#include <string>

int status;
const char *getInput()
{
    return status == 0 ? std::getenv("TMP") : nullptr;

}

void foo()
{
    status=1;
    const char *data = getInput();
    //...
    std::string str(data);   // Noncompliant
    str.append(data);        // Noncompliant
    str.assign(data);        // Noncompliant
    str.insert(0, data);     // Noncompliant
    str.replace(0, 1, data); // Noncompliant
}
```

In this example, the `const char*` object `data` is created by calling `getInput()`, and then various string operations are performed by using `data`. Polyspace reports a violation of this rule for each string operation because the function `getInput()` returns a `nullptr` which is then assigned to `data`.

#### Correction — Avoid Performing String Operation on Null Pointers

Modify `getInput()` so that the function does not return a `nullptr`.

```
#include <cstdlib>
#include <string>

int status;
const char *getInput()
{
    return status == 0 ? std::getenv("TMP") : "";

}

void foo()
{
    status=1;
    const char *data = getInput();
    //...
    std::string str(data);   // Compliant
    str.append(data);        // Compliant
    str.assign(data);        // Compliant
    str.insert(0, data);     // Compliant
    str.replace(0, 1, data); // Compliant
}
```

## Check Information

**Group:** 05. Characters and Strings (STR)

# Version History

**Introduced in R2022b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

STR51-CPP

# CERT C++: STR52-CPP

Use valid references, pointers, and iterators to reference elements of a `basic_string`

## Description

### Rule Definition

*Use valid references, pointers, and iterators to reference elements of a `basic_string`.*

### Polyspace Implementation

The rule checker checks for the issue **Use of invalid string iterator**.

## Examples

### Use of Invalid String Iterator

#### Issue

This issue occurs when you use an iterator, pointer, or reference to an element of a `basic_string` object that has been invalidated by actions such as insertion or erasure. This issue is a specific instance of the issue **Use of invalid iterator**, which causes violations of `CERT C++: CTR51-CPP`. Consider this code:

```
std::string str = "Basic String";
auto it_begin = str.begin();
//...
str.replace(0,5,"Advanced");
str.insert(it_begin,'A'); // Violation
```

Here, the iterator `it_begin` is invalidated after the call to `string::replace`. Using this invalid iterator in `string::insert` causes a violation of this rule.

Functions that invalidate an STL iterator also invalidate `basic_string` iterators. In addition, these functions invalidate a `basic_string` iterator:

- `std::swap()`
- `Operator>>(std::basic_istream&, std::string&)`
- `std::getline()`

#### Risk

Dereferencing the invalid iterator might produce unexpected results. If the invalidated iterator becomes a dangling pointer or an uninitialized pointer, dereferencing it might cause sudden termination of the program.

#### Fix

The C++ standard defines which operations invalidate the iterators of the standard library containers. These operations also invalidate a string operator. See Containers library. When you perform actions that might invalidate an iterator, revalidate the iterator. For instance, recalculate the iterator after an insertion or erasure.

#### Example — Avoid Using Invalid Iterators

```
#include <string>


void foo(const std::string& raw, std::string& processed) {

  // Process the raw string and store in processed
  auto loc = processed.begin();
  for (auto i = raw.begin(); i != raw.end(); ++i, ++loc) {
    processed.insert(loc, *i != ';' ? *i : ' ');//Noncompliant
  }
}
```

In this example, the first `processed.insert` operation in the `for` loop invalidates the iterator `i`. The subsequent use of the iterator `i` might result in undefined behavior. Polyspace reports a violation of this rule.

#### Correction — Update the Iterator

When you use iterators after an insertion or erasure, update the iterator to avoid undefined behavior. For instance, in this code, the iterator `i` is updated after each insertion so that the iterator remains valid.

```
#include <string>


void foo(const std::string& raw, std::string& processed) {

  // Process the raw string and store in processed
  auto loc = processed.begin();
  for (auto i = raw.begin(); i != raw.end(); ++i, ++loc) {
    loc = processed.insert(loc, *i != ';' ? *i : ' ');//Compliant
  }
}
```

## Check Information

**Group:** 05. Characters and Strings (STR)

# Version History

**Introduced in R2022b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

STR52-CPP

# CERT C++: STR53-CPP

Range check element access

## Description

### Rule Definition

*Range check element access.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access out of bounds**.
- **Array access with tainted index**.
- **Pointer dereference with tainted offset**.

### Extend Checker

Extend this checker to run a stricter analysis:

- When the input values are unknown and only a subset of inputs causes an issue, a default Bug Finder analysis might not detect an **Array access out of bounds**. To check for this issue when it is caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.
- A default Bug Finder analysis might not flag a **Array access with tainted index** or **Pointer dereference with tainted offset** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Array access out of bounds

#### Issue

**Array access out of bounds** occurs when an array index falls outside the range `[0...array_size-1]` during array access.

#### Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.

- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example - Array Access Out of Bounds Error**

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);    //Noncompliant
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

**Correction — Keep Array Index Within Array Bounds**

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

### Array access with tainted index

#### Issue

**Array access with tainted index** detects reading or writing to an array by using a tainted index that has not been validated.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

#### Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

#### Example - Use Index to Return Buffer Value

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];   //Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The index `num` is obtained from an unsecure source and the function `taintedarrayindex` does not check to see if `num` is inside the range of `tab`.

**Correction — Check Range Before Use**

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

### Pointer dereference with tainted offset

**Issue**

**Pointer dereference with tainted offset** detects pointer dereferencing, either reading or writing, using an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

**Risk**

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

**Fix**

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

**Example - Dereference Pointer Array**

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);
```

```
int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset]; //Noncompliant
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of `offset`. Continue with the pointer dereferencing only if `offset` is inside the valid range.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

## Check Information

**Group:** 05. Characters and Strings (STR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

STR53-CPP

# CERT C++: MEM30-C

Do not access freed memory

## Description

### Rule Definition

*Do not access freed memory.*

### Polyspace Implementation

This checker checks for:

- **Accessing previously freed pointer**
- **Freeing previously freed pointer**

## Examples

### Accessing previously freed pointer

#### Issue

**Accessing previously freed pointer** occurs when you attempt to access a block of memory after freeing the block by using the `free` function.

#### Risk

When a pointer is allocated dynamic memory by using `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation and the pointer becomes a dangling pointer. Attempting to access this block of memory by dereferencing the dangling pointer can result in unpredictable behavior or a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. Determine if you intended to free the memory later or allocate another memory block to the pointer before access.

As a best practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

#### Example — Accessing Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   { 
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift; //Noncompliant

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing `pi` after the `free` statement is not valid.

#### Correction — Free Pointer After Last Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

### Freeing previously freed pointer

#### Issue

**Freeing previously freed pointer** occurs when you attempt to free the memory allocated to a pointer after already freeing the pointer by using the `free` function.

#### Risk

Attempting to free the memory associated with a previously freed pointer might corrupt the memory management of the program and cause a memory leak. This defect might allow an attacker to access the memory and execute arbitrary code.

#### Fix

To avoid this defect, assign pointers to NULL after freeing them. Check the pointers for NULL value before attempting to access the memory associated with the pointer. In this way, you are protected against accessing a freed block.

#### Example — Freeing Previously Freed Pointer

```
#include <stdlib.h>
#include <stdio.h>
int getStatus();
void double_deallocation(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == 0) return;

    *pi = 2;
    /*...*/
    if(getStatus()==1)
    {
        /*...*/
        free(pi);
    }
    free(pi); //Noncompliant
}
```

The second `free` statement attempts to release the block of memory that `pi` refers to, but the pointer `pi` might already be freed in the `if` block of code. This second `free` statement might cause a memory leak and security vulnerabilities in the code. Polyspace flags the second `free` statement.

**Correction — Check Pointers Before Calling `free`**

One possible correction is to assign freed pointers to NULL and to check pointers for NULL before freeing them.

```
#include <stdlib.h>
#include <stdio.h>
int getStatus();
void double_deallocation(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == 0) return;

    *pi = 2;
    /*...*/
    if(getStatus()==1)
    {
        /*...*/
        if(pi!=NULL)
        {
            free(pi);
            pi= NULL;
        }
    }
    /*...*/
    if(pi!=NULL)
    {
        free(pi);
        pi= NULL;
    } //Compliant
}
```

In this case, the memory allocated to pointer `pi` is freed only if it is not already freed.

## Check Information

**Group:** 06. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MEM30-C

# CERT C++: MEM31-C

Free dynamically allocated memory when no longer needed

## Description

### Rule Definition

*Free dynamically allocated memory when no longer needed.*

### Polyspace Implementation

The rule checker checks for **Memory leak**.

## Examples

### Memory leak

#### Issue

**Memory leak** occurs when you do not free a block of memory allocated through `malloc`, `calloc`, `realloc`, or `new`. If the memory is allocated in a function, the defect does not occur if:

- Within the function, you free the memory using `free` or `delete`.
- The function returns the pointer assigned by `malloc`, `calloc`, `realloc`, or `new`.
- The function stores the pointer in a global variable or in a parameter.

#### Risk

Dynamic memory allocation functions such as `malloc` allocate memory on the heap. If you do not release the memory after use, you reduce the amount of memory available for another allocation. On embedded systems with limited memory, you might end up exhausting available heap memory even during program execution.

#### Fix

Determine the scope where the dynamically allocated memory is accessed. Free the memory block at the end of this scope.

To free a block of memory, use the `free` function on the pointer that was used during memory allocation. For instance:

```
ptr = (int*)malloc(sizeof(int));
...
free(ptr);
```

It is a good practice to allocate and free memory in the same module at the same level of abstraction. For instance, in this example, `func` allocates and frees memory at the same level but `func2` does not.

```
void func() {
  ptr = (int*)malloc(sizeof(int));
  {
    ...
  }
  free(ptr);
}

void func2() {
  {
   ptr = (int*)malloc(sizeof(int));
   ...
  }
  free(ptr);
}
```

See CERT-C Rule MEM00-C.

#### Example - Dynamic Memory Not Released Before End of Function

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
} //Noncompliant
```

In this example, `pi` is dynamically allocated by `malloc`. The function `assign_memory` does not free the memory, nor does it return `pi`.

#### Correction — Free Memory

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
    free(pi);
}
```

#### Correction — Return Pointer from Dynamic Allocation

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

```
#include<stdlib.h>
#include<stdio.h>

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
            printf("Memory allocation failed");
            return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

#### Example - Memory Leak with New/Delete

```
#define NULL '\0'

void initialize_arr1(void)
{
    int *p_scalar = new int(5);
} //Noncompliant

void initialize_arr2(void)
{
    int *p_array = new int[5];
} //Noncompliant
```

In this example, the functions create two variables, `p_scalar` and `p_array`, using the `new` keyword. However, the functions end without cleaning up the memory for these pointers. Because the functions used `new` to create these variables, you must clean up their memory by calling `delete` at the end of each function.

#### Correction — Add Delete

To correct this error, add a `delete` statement for every `new` initialization. If you used brackets `[]` to instantiate a variable, you must call delete with brackets as well.

```
#define NULL '\0'

void initialize_arrs(void)
{
    int *p_scalar = new int(5);
    int *p_array = new int[5];

    delete p_scalar;
    p_scalar = NULL;
    delete[] p_array;
    p_scalar = NULL;
}
```

## Check Information

**Group:** 06. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MEM31-C

# CERT C++: MEM34-C

Only free memory allocated dynamically

## Description

### Rule Definition

*Only free memory allocated dynamically.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Invalid free of pointer**
- **Invalid reallocation of pointer**

## Examples

### Invalid free of pointer

#### Issue

**Invalid free of pointer** occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

#### Risk

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function or a previous `malloc` function on a different pointer.

#### Fix

In most cases, you can fix the issue by removing the `free` statement. If the pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

#### Example - Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
```

```
     *(p+i)=1;

  free(p);    //Noncompliant
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is deallocated using the `free` function. However, `p` points to a memory location that was not dynamically allocated.

#### Correction — Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

#### Correction — Introduce Pointer Allocation

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p`.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

### Invalid reallocation of pointer

#### Issue

**Invalid reallocation of pointer** occurs when a block of memory reallocated using the `realloc` function was not previously allocated using `malloc` or `calloc`.

#### Risk

Reallocating a block of memory that was not allocated dynamically allocated can result in undefined behavior.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `realloc` function on a different pointer.

**Fix**

If you want to reallocate a block of memory, make sure that it was dynamically allocated in the first place.

**Example – Reallocation of Memory Allocated Statically**

```
#include <cstdlib>

#define SIZE 256

void reshape(int isSpaceAvailable) {
  char buf[SIZE];
  char *newBuf;
  newBuf = (char *)realloc(buf, 2 * SIZE); //Noncompliant

  if (newBuf == NULL) {
    /* Handle error */
  }
}
```

In this example, the buffer `buf` is not allocated dynamically. Therefore, the reallocation of `buf` results in undefined behavior.

**Example – Reallocation of Memory Allocated Statically**

Make sure that a buffer that you reallocate was previous allocated memory dynamically.

```
#include <cstdlib>

#define SIZE 256

void reshape(void){
  char *buf;
  char *newBuf;
  buf = (char*)malloc(SIZE *sizeof(char));
  newBuf = (char *)realloc(buf, 2 * SIZE);

  if (newBuf == NULL) {
    /* Handle error */
  }
  free(newBuf);
}
```

## Check Information

**Group:** 06. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

MEM34-C

# CERT C++: MEM35-C

Allocate sufficient memory for an object

## Description

### Rule Definition

*Allocate sufficient memory for an object.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Pointer access out of bounds**.
- **Memory allocation with tainted size**.
- **Insufficient memory allocation**

### Extend Checker

A default Bug Finder analysis might not flag a **Memory allocation with tainted size** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Pointer access out of bounds

#### Issue

**Pointer access out of bounds** occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

#### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example — Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i; //Noncompliant
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

#### Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

### Memory allocation with tainted size

#### Issue

**Memory allocation with tainted size** checks memory allocation functions, such as `calloc` or `malloc`, for size arguments from unsecured sources.

#### Risk

Uncontrolled memory allocation can cause your program to request too much system memory. This consequence can lead to a crash due to an out-of-memory condition, or assigning too many resources.

#### Fix

Before allocating memory, check the value of your arguments to check that they do not exceed the bounds.

#### Example — Allocate Memory Using Input Argument

```
#include<stdio.h>
#include <stdlib.h>

int* bug_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = (int*)malloc(size); //Noncompliant
    return p;
}
```

In this example, `malloc` allocates `size` amount of memory for the pointer `p`. `size` comes from the user of the program and its value is not checked. If the size is larger than the amount of memory you have available, your program could crash.

#### Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the `malloc` operation. This example checks to see if the size is positive and less than the maximum size.

```
#include "stdlib.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int* corrected_taintedmemoryallocsize(int size) {
    int* p = NULL;
          /* Fix: Check entry range before use */
    if (size>0 && size<SIZE128) {
        p = (int*)malloc((unsigned int)size);
    }
    return p;
}
```

### Insufficient memory allocation

#### Issue

**Insufficient memory allocation** occurs when the allocated memory is not sufficient to hold the target object. Consider this code:

```
#include <stdlib.h>

typedef struct S {
    int a;
    int b;
    int c;
}S;

void foo(){
    S* a;
    a = (S*)malloc(sizeof(S*)*2);//Noncompliant
}
void bar(){
    S* a;
    a = (S*)malloc(sizeof(S*)*3); //Compliant
}
```

Objects of type `S` must accommodate three integers, which typically requires 12 bytes of memory. In `foo()` the allocated memory size for the `S` type object is eight bytes. Because the allocated memory is insufficient to hold the target object, Polyspace reports a violation of this rule. In `bar()`, the allocated memory is 12 bytes, which is sufficient. Polyspace does not report a violation.

#### Risk

Insufficient memory allocation results in buffer overflow and unexpected termination of the program.

#### Fix

When allocating memory blocks, allocate sufficient memory.

#### Example — Allocate Memory for Objects by Calling `malloc()`

In this example, the function `foo()` allocates memory for pointers `a` and `b`:

- When allocating memory for `a`, the argument of `sizeof()` is `S*`, which is a pointer. In a 64 bit system, all pointer types are 8 bytes in size. The `malloc()` statement allocates 24 bytes, which is sufficient to accommodate an object of `S` type. Because the allocated memory can hold the `a` object, Polyspace does not report a violation. This statement causes a `Wrong type used in sizeof` defect.

  From the context of the `malloc()` statement, the object `a` is intended to be a three element array of `S` type objects. Clearly, the `malloc()` statement does not allocate sufficient memory for the entire array. In the `for` loop, when you access the elements of the array, the insufficient memory causes a violation of this rule. This way, the `Wrong type used in sizeof` defect in the `malloc()` statement causes a violation of this rule when you use `a` as an array. If you use `a` as an object instead of an array, this rule is not violated.
- When allocating memory for `b`, the `malloc()` statement allocates enough memory to accommodate two integers. Because objects of type `S` require enough memory to accommodate three integers, the allocated memory is insufficient. Polyspace reports a violation of this rule.

```
#include <stdlib.h>


typedef struct S
{
  int a;
  int b;
  int c;
} S;

void setS (S * a, int x, int y, int z)
{
  a->a = x;            //Noncompliant: pointer access out of bounds
  a->b = y;
  a->c = z;
}

void foo ()
{
  S *a, *b;
  a = (S *) malloc (3 * sizeof (S *));    //Compliant but causes PTR_SIZEOF_MISMATCH defect/
  setS(a,22,32,45); //Compliant
  for (int i = 1; i < 3; ++i)
    {
      setS (a + i, i, i * i, i * i * i);

    }
  b = (S *) malloc (2 * sizeof (int));    //Noncompliant

}
```

**Correction — Allocate Sufficient Memory**

To fix this issue, allocate sufficient memory. The best practice is to use the `sizeof()` function with correct argument type and a correct multiplier to calculate how much memory is required.

```
#include <stdlib.h>


typedef struct S
{
  int a;
  int b;
  int c;
} S;

void setS (S * a, int x, int y, int z)
{
  a->a = x;            //Compliant: pointer access out of bounds
  a->b = y;
  a->c = z;
}

void foo ()
{
  S *a, *b;
  a = (S *) malloc (3 * sizeof (S));    //No violations of PTR_SIZEOF_MISMATCH*/
  setS(a,22,32,45); //Compliant
  for (int i = 1; i < 3; ++i)
    {
      setS (a + i, i, i * i, i * i * i);

    }
  b = (S *) malloc (1 * sizeof (S));    //Compliant
  //...
  free(a);
  free (b);
}
```

## Check Information

**Group:** 06. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C++ (-cert-cpp))

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MEM35-C

# CERT C++: MEM36-C

Do not modify the alignment of objects by calling realloc()

## Description

### Rule Definition

*Do not modify the alignment of objects by calling realloc().*

### Polyspace Implementation

The rule checker checks for **Alignment changed after memory reallocation**.

## Examples

### Alignment changed after memory reallocation

#### Issue

**Alignment changed after memory reallocation** occurs when you use `realloc()` to modify the size of objects with strict memory alignment requirements.

#### Risk

The pointer returned by `realloc()` can be suitably assigned to objects with less strict alignment requirements. A misaligned memory allocation can lead to buffer underflow or overflow, an illegally dereferenced pointer, or access to arbitrary memory locations. In processors that support misaligned memory, the allocation impacts the performance of the system.

#### Fix

To reallocate memory:

1. Resize the memory block.
   - In Windows, use `_aligned_realloc()` with the alignment argument used in `_aligned_malloc()` to allocate the original memory block.
   - In UNIX/Linux, use the same function with the same alignment argument used to allocate the original memory block.
2. Copy the original content to the new memory block.
3. Free the original memory block.

**Note** This fix has implementation-defined behavior. The implementation might not support the requested memory alignment and can have additional constraints for the size of the new memory.

#### Example - Memory Reallocated Without Preserving the Original Alignment

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define SIZE1024 1024

void func(void)
{
    size_t resize = SIZE1024;
    size_t alignment = 1 << 12; /* 4096 bytes alignment */
    int *ptr = NULL;
    int *ptr1;

    /* Allocate memory with 4096 bytes alignment */

    if (posix_memalign((void **)&ptr, alignment, sizeof(int)) != 0)
    {
        /* Handle error */
      }

    /*Reallocate memory without using the original alignment.
    ptr1 may not be 4096 bytes aligned. */

    ptr1 = (int *)realloc(ptr, sizeof(int) * resize); //Noncompliant

    if (ptr1 == NULL)
    {
        /* Handle error */
    }

    /* Processing using ptr1 */

    /* Free before exit */
    free(ptr1);
}
```

In this example, the allocated memory is 4096-bytes aligned. `realloc()` then resizes the allocated memory. The new pointer `ptr1` might not be 4096-bytes aligned.

**Correction — Specify the Alignment for the Reallocated Memory**

When you reallocate the memory, use `posix_memalign()` and pass the alignment argument that you used to allocate the original memory.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE1024 1024

void func(void)
{
    size_t resize = SIZE1024;
    size_t alignment = 1 << 12; /* 4096 bytes alignment */
    int *ptr = NULL;

    /* Allocate memory with 4096 bytes alignment */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int)) != 0)
    {
        /* Handle error */
    }

    /* Reallocate memory using the original alignment. */
    if (posix_memalign((void **)&ptr, alignment, sizeof(int) * resize) != 0)
    {
        /* Handle error */
        free(ptr);
        ptr = NULL;
    }

    /* Processing using ptr */

    /* Free before exit */
    free(ptr);
}
```

## Check Information

**Group:** 06. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MEM36-C

# CERT C++: MEM50-CPP

Do not access freed memory

## Description

### Rule Definition

*Do not access freed memory.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Pointer access out of bounds**.
- **Deallocation of previously deallocated pointer**.
- **Use of previously freed pointer**.

## Examples

### Pointer access out of bounds

#### Issue

**Pointer access out of bounds** occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

#### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example - Pointer access out of bounds error**

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i; //Noncompliant
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

**Correction — Check Pointer Stays Within Bounds**

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

### Deallocation of previously deallocated pointer

**Issue**

**Deallocation of previously deallocated pointer** occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

#### Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to free this block of memory can result in a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. See if you intended to allocate a memory block to the pointer between the first deallocation and the second. Otherwise, remove the second `free` statement.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before freeing pointers, check them for NULL values and handle the error. In this way, you are protected against freeing an already freed block.

#### Example - Deallocation of Previously Deallocated Pointer Error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);        //Noncompliant
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

#### Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
 }
```

### Use of previously freed pointer

#### Issue

**Use of previously freed pointer** occurs when you access a block of memory after freeing the block using the `free` function.

**Risk**

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

**Fix**

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

**Example - Use of Previously Freed Pointer Error**

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift; //Noncompliant
    /* Defect: Reading a freed pointer */

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

**Correction — Free Pointer After Use**

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

## Check Information

**Group:** 06. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MEM50-CPP

# CERT C++: MEM51-CPP

Properly deallocate dynamically allocated resources

## Description

### Rule Definition

*Properly deallocate dynamically allocated resources.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Invalid deletion of pointer**.
- **Invalid free of pointer**.
- **Deallocation of previously deallocated pointer**.

## Examples

### Invalid deletion of pointer

#### Issue

**Invalid deletion of pointer** occurs when:

- You release a block of memory with the `delete` operator but the memory was previously not allocated with the `new` operator.
- You release a block of memory with the `delete` operator using the single-object notation but the memory was previously allocated as an array with the `new` operator.

This defect applies only to C++ source files.

#### Risk

The risk depends on the cause of the issue:

- The `delete` operator releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.
- If you use the single-object notation for `delete` on a pointer that is previously allocated with the array notation for `new`, the behavior is undefined.

The issue can also highlight other coding errors. For instance, you perhaps wanted to use the `delete` operator or a previous `new` operator on a different pointer.

#### Fix

The fix depends on the cause of the issue:

- In most cases, you can fix the issue by removing the `delete` statement. If the pointer is not allocated memory from the heap with the `new` operator, you do not need to release the pointer with `delete`. You can simply reuse the pointer as required or let the object be destroyed at the end of its scope.

- In case of mismatched notation for `new` and `delete`, correct the mismatch. For instance, to allocate and deallocate a single object, use this notation:

  ```
  classType* ptr = new classType;
  delete ptr;
  ```

  To allocate and deallocate an array objects, use this notation:

  ```
  classType* p2 = new classType[10];
  delete[] p2;
  ```

If the issue highlights a coding error such as use of `delete` or `new` on the wrong pointer, correct the error.

**Example - Deleting Static Memory**

```
void assign_ones(void)
{
    int ptr[10];

    for(int i=0;i<10;i++)
        *(ptr+i)=1;

    delete[] ptr;    //Noncompliant
}
```

The pointer `ptr` is released using the `delete` operator. However, `ptr` points to a memory location that was not dynamically allocated.

**Correction: Remove Pointer Deallocation**

If the number of elements of the array `ptr` is known at compile time, one possible correction is to remove the deallocation of the pointer `ptr`.

```
void assign_ones(void)
{
    int ptr[10];

    for(int i=0;i<10;i++)
        *(ptr+i)=1;
}
```

**Correction — Add Pointer Allocation**

If the number of array elements is not known at compile time, one possible correction is to dynamically allocate memory to the array `ptr` using the `new` operator.

```
void assign_ones(int num)
{
    int *ptr = new int[num];

    for(int i=0; i < num; i++)
        *(ptr+i) = 1;

    delete[] ptr;
   }
```

**Example - Mismatched `new` and `delete`**

```
int main (void)
{
```

```
    int *p_scale = new int[5];

    //more code using scal

    delete p_scale; //Noncompliant
}
```

In this example, `p_scale` is initialized to an array of size 5 using `new int[5]`. However, `p_scale` is deleted with `delete` instead of `delete[]`. The `new`-`delete` pair does not match. Do not use `delete` without the brackets when deleting arrays.

#### Correction — Match delete to new

One possible correction is to add brackets so the `delete` matches the `new []` declaration.

```
int main (void)
{
    int *p_scale = new int[5];

    //more code using p_scale

    delete[] p_scale;
}
```

#### Correction — Match new to delete

Another possible correction is to change the declaration of `p_scale`. If you meant to initialize `p_scale` as 5 itself instead of an array of size 5, you must use different syntax. For this correction, change the square brackets in the initialization to parentheses. Leave the `delete` statement as it is.

```
int main (void)
{
    int *p_scale = new int(5);

    //more code using p_scale

    delete p_scale;
}
```

### Invalid free of pointer

#### Issue

**Invalid free of pointer** occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

#### Risk

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function or a previous `malloc` function on a different pointer.

#### Fix

In most cases, you can fix the issue by removing the `free` statement. If the pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

### Example - Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);    //Noncompliant
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is deallocated using the `free` function. However, `p` points to a memory location that was not dynamically allocated.

### Correction — Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

### Correction — Introduce Pointer Allocation

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p`.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

### Deallocation of previously deallocated pointer

#### Issue

**Deallocation of previously deallocated pointer** occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

#### Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to free this block of memory can result in a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. See if you intended to allocate a memory block to the pointer between the first deallocation and the second. Otherwise, remove the second `free` statement.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before freeing pointers, check them for NULL values and handle the error. In this way, you are protected against freeing an already freed block.

#### Example - Deallocation of Previously Deallocated Pointer Error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);        //Noncompliant
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

#### Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
 }
```

## Check Information

**Group:** 06. Memory Management (MEM)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MEM51-CPP

# CERT C++: MEM52-CPP

Detect and handle memory allocation errors

## Description

### Rule Definition

*Detect and handle memory allocation errors.*

### Polyspace Implementation

The rule checker checks for **Unprotected dynamic memory allocation**.

## Examples

### Unprotected dynamic memory allocation

#### Issue

**Unprotected dynamic memory allocation** occurs when you do not check after dynamic memory allocation whether the memory allocation succeeded.

#### Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

#### Fix

Check the return value of `malloc`, `calloc`, or `realloc` for NULL before accessing the allocated memory location.

```
int *ptr = malloc(size * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

#### Example - Unprotected dynamic memory allocation error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));

  *p = 2;   //Noncompliant
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function `calloc` returns `NULL` to `p`. Before accessing the memory through `p`, the code does not check whether `p` is `NULL`

#### Correction — Check for NULL Value

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

## Check Information

**Group:** 06. Memory Management (MEM)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MEM52-CPP

# CERT C++: MEM53-CPP

Explicitly construct and destruct objects when manually managing object lifetime

## Description

### Rule Definition

*Explicitly construct and destruct objects when manually managing object lifetime.*

### Polyspace Implementation

The rule checker checks for **Object allocated but not initialized**. Polyspace does not look for explicit destruction of manually managed objects.

## Examples

### Object Allocated But Not Initialized

#### Issue

This issue occurs when a nontrivially constructible object is created by calling functions such as `std::malloc()` or `std::allocator<T>::allocate()`. These functions allocate memory for objects but do not initialize them. To check if a class is trivially constructible, use the function `std::is_trivially_constructible`. See https://en.cppreference.com/w/cpp/types/is_constructible.

Polyspace checks for this issue in classes that has a defined constructor. If a class lacks a defined constructor, Polyspace does not report violations of this rule.

#### Risk

Calling the function `malloc()` or `std::allocator<T>::allocate()` allocates the memory necessary for an object, but does not start the lifetime of the allocated object. Using these functions to construct an object might cause you to access class members before their lifetime starts. Accessing objects before their lifetime is an undefined behavior. For instance:

```
class myObj {
    myObj();

    void myFunc();
};

void foo() {
    //Allocating sufficient memory
    myObj* obj = static_cast<myObj*>(std::malloc(sizeof(myObj)));
    // Accessing myObj::myFunc
    obj->myFunc();


}
```

The code allocates sufficient memory for the object `obj` by calling the function `std::malloc()`. The function `malloc()` does not initiate the lifetime of `obj` and its members. In `obj->myFunc`, you access `muFunc()` before its lifetime, which is an undefined behavior.

**Fix**

When using the function `std::malloc()` or `std::allocator<T>::allocate()` to construct an object, explicitly complete the construction by using the placement-`new` operator. This step starts the lifetime of the object and prevent the undefined behavior resulting from accessing objects before their lifetime.

**Example — Accessing Objects Before Lifetime**

```
#include <memory>
template <typename T, typename Allocator = std::allocator<T>>
class customManager{
    T* memChunk;
public:
    void Allocate(){
        Allocator alloc;
        T* p = alloc.allocate(1);
        memChunk = p;
    }

    void Deallocate(){
        //...
    }
    T* access(){
        //..
        return memChunk;
    }
};

class myClass{
public:
    myClass(){}
    int num;
    void getNum(){};//Noncompliant

};
void foo(){
    customManager<myClass> managedObj;
    managedObj.Allocate();
    (managedObj.access())->getNum();}
```

In this example, the template `customManager` manages memory for a class. In the function `foo`, this template is used to manage a `myClass` object. The function `CustomManager::Allocate` allocates memory but does not initiate the lifetime of its client objects. Accessing `myClass::getnum()` after allocating `managedObj` results in accessing an object before its lifetime, which is undefined behavior. Polyspace reports a violation of this rule.

**Correction — Explicitly Initiate Object Lifetime**

Explicitly start the lifetime of the objects after allocating memory for them. For instance, after calling `alloc.allocate()`, initiate the lifetime of the client objects by calling the placement `new` operator.

```
#include <memory>
template <typename T, typename Allocator = std::allocator<T>>
```

```
class customManager{
    T* memChunk;
public:
    void Allocate(){
        Allocator alloc;
        T* p = alloc.allocate(1);
        memChunk = new (p) T;//Lifetime of the object starts
    }

    void Deallocate(){
        //...
    }
    T* access(){
        //..
        return memChunk;
    }
};

class myClass{
public:
    myClass(){}
    int num;
    void getNum(){};
};
void foo(){
    customManager<myClass> managedObj;
    managedObj.Allocate();
    (managedObj.access())->getNum();
}
```

## Check Information

**Group:** 06. Memory Management (MEM)

# Version History

**Introduced in R2022b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MEM53-CPP

# CERT C++: MEM54-CPP

Provide placement new with properly aligned pointers to sufficient storage capacity

## Description

### Rule Definition

*Provide placement new with properly aligned pointers to sufficient storage capacity*

### Polyspace Implementation

The rule checker checks for **Placement new used with insufficient storage or misaligned pointers**.

## Examples

### Placement new used with insufficient storage or misaligned pointers

#### Issue

**Placement new used with insufficient storage or misaligned pointers** occurs when the pointer passed to a placement `new` operator does not have sufficient storage for the memory allocation or is not properly aligned.

Suppose that a pointer `ptr` is preallocated `m` bytes of memory on the stack and has alignment `n`. For instance, if `ptr` is an array:

```
uint8_t ptr[5];
```

the allocated storage is `sizeof(uint8_t) * 5` and the alignment is `alignof(uint8_t)`. If you allocate more than `m` bytes to this pointer in a placement `new` expression or if the alignment required for the allocation is greater than `n`, the checker raises a violation. When determining the pointer alignment, the checker takes into account explicit alignments such as with `std::align`.

The checker does not consider pointers that are preallocated memory on the heap since the available storage depends on the memory availability, which is known only at run time.

#### Risk

The `new` operator allocates the required amount of memory for storing an object on the heap and constructs a new object in the allocated memory in a single operation. If you want to separate the allocation and the construction and place an object in preallocated memory on either the stack or the heap, you use placement `new`. Placement `new` has advantages over `new` in certain situations, for example, when you need to place the object at a known memory location.

The `new` operator automatically allocates the correct amount of aligned memory that the object requires. But when using placement `new`, you must manually make sure that the pointer you pass has sufficient allocated storage capacity and is properly aligned. Violating these constraints results in the construction of an object at a misaligned location or memory initialization outside of allocated bounds, which might lead to unexpected or implementation-dependent behavior.

#### Fix

Make sure that the pointer used in the placement new operation has sufficient memory for the allocation and the alignments match.

**Example — Placement new Used with Insufficient Storage Capacity and Misaligned Pointers**

```
#include <new>
#include<memory>
#include <cstdint>

void Foo()
{
  uint8_t c;
  uint64_t* ptr =
      new    // Non-compliant (insufficient storage, misaligned)
      (&c) uint64_t;
}

void Bar()
{
  uint8_t buf[sizeof(uint64_t)];
  uint64_t* ptr =
      new            // Non-compliant (sufficient storage, misaligned)
      (buf) uint64_t;
}

void Baz()
{
  void* buf;
  std::size_t sp = 64;
  std::align(alignof(uint64_t), sizeof(uint64_t), buf, sp);
  uint64_t* ptr =
      new            // Compliant (sufficient storage, aligned)
      (buf) uint64_t;
}
```

In the function Foo, the &c points to an uint8_t value and has one byte memory in stack with one-byte alignment. The pointer is passed to placement new, which constructs an instance of uint64_t that requires 8 bytes of memory and a 4-byte alignment. This usage violates the rule.

In the function Bar, the pointer buf is properly allocated and has sufficient storage capacity. But, because it points to the uint8_t data type, it has one-byte alignment. This usage still violates the rule.

The function Baz calls the std::align function to create a pointer with correct storage capacity (8 byte) and alignment (4-byte) for uint64_t. This usage complies with the rule.

## Check Information

**Group:** 06. Memory Management (MEM)

# Version History

**Introduced in R2020b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

MEM54-CPP

# CERT C++: MEM55-CPP

Honor replacement dynamic storage management requirements

## Description

### Rule Definition

*Honor replacement dynamic storage management requirements.*

### Polyspace Implementation

The rule checker checks for **Replacement allocation/deallocation functions that do not meet requirements of the Standard**.

## Examples

### Replacement allocation/deallocation functions that do not meet requirements of the Standard

#### Issue

The issue occurs when you provide these replacement implementations of dynamic allocation and deallocation functions in the global namespace:

- Replacement `operator new` that returns `nullptr`.

  The expected behavior is to throw a `bad_alloc` exception on failure.
- Replacement `operator new` or `operator delete` that throws directly or indirectly on failure.

  The expected behavior is that dynamic allocation or deallocation functions must not throw. Polyspace also highlights the location of the throw in your code.

#### Risk

The C++ Standard (*[new.delete]*) specifies certain required behaviors for the dynamic allocation and deallocation functions. If you implement a global replacement allocation or deallocation function that does not meet these semantic requirements, other functions that rely on the required behaviors might behave in an undefined manner.

For instance, `void* operator new ( std::size_t count )` is expected to throw a `bad_alloc` exception if it fails to allocate the requested amount of memory. If you implement a replacement allocation function that returns `nullptr` instead of throwing, a function that expect the memory allocation to throw on failure might try to dereference a null pointer instead.

#### Fix

If you provide global replacements for dynamic allocation/deallocation functions, implement the semantic requirements specified by the standard in the corresponding required behavior paragraphs.

#### Non-Throwing `operator new` That Throws

```
#include<cstdlib>
#include<new>
```

```
extern void* custom_alloc(std::size_t);

void* operator new (std::size_t count, const std::nothrow_t& tag) //Non-compliant
{
    if (void* ret = custom_alloc(count)) {
        return ret;
    }
    throw std::bad_alloc();
}

void func()
{
    int* ptr1 = new int;
    if (ptr1) {
        //Use ptr1
    }

}
```

In this example, the replacement dynamic allocation function is specified as non-throwing (`std::nothrow_t`) but throws a `bad_alloc` exception on failure. Function `func`, which expects `operator new` to return a null pointer on failure, does not handle the exception and this might result in an abrupt program termination.

#### Correction

One possible correction is to provide a replacement `operator new` function that does not throw.

```
#include<cstdlib>
#include<new>

extern void* custom_alloc(std::size_t);

void* operator new (std::size_t count, const std::nothrow_t& tag)
{
    return custom_alloc(count);
}

void func()
{
    int* ptr1 = new int;
    if (ptr1) {
        //Use ptr1
    }

}
```

## Check Information

**Group:** 06. Memory Management (MEM)

# Version History

**Introduced in R2020b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MEM55-CPP

# CERT C++: MEM56-CPP

Do not store an already-owned pointer value in an unrelated smart pointer

## Description

### Rule Definition

*Do not store an already-owned pointer value in an unrelated smart pointer.*[18]

### Polyspace Implementation

The rule checker checks for **Use of already-owned pointers**

## Examples

### Use of already-owned pointers

#### Issue

This issue occurs when you use an already-owned pointer as the argument of:

- A smart pointer constructor. For instance, in this code snippet, `raw_ptr` is already owned by `s_ptr1` and is used to initialize `s_ptr2`:

```
char *raw_ptr = new char;
std::shared_ptr<char> s_ptr1(raw_ptr);
std::shared_ptr<char> s_ptr2(raw_ptr); //raw_ptr is already owned by s_ptr1
```

- A smart pointer reset operation. For instance, in this code snippet, the reset of `s_ptr2` replaces `raw_ptr2` with already-owned `raw_ptr1`:

```
char *raw_ptr1 = new char;
char *raw_ptr2 = new char;

std::shared_ptr<char> s_ptr1(raw_ptr1);
std::shared_ptr<char> s_ptr2(raw_ptr2);

s_ptr2.reset(raw_ptr1); // s_ptr2 releases raw_ptr2 and owns already owned raw_ptr1
```

Polyspace checks only smart pointer types `std::shared_ptr` and `std::unique_ptr` and considers that user-defined allocators and deleters have standard allocation and deallocation behavior.

A pointer is already owned by a smart pointer if the pointer type is not `std::nullptr_t` and either:

[18] *This software has been created by MathWorks incorporating portions of: the "SEI CERT-C Website," © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, "SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition," © 2016 Carnegie Mellon University, and "SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition" © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

- The pointer was used to initialize the smart pointer.
- The pointer was used as an argument to the smart pointer `reset()` member function.
- The pointer is the return value of the smart pointer `get()` member function.
- The pointer is the return value of the smart pointer `operator->` member function.

**Risk**

You use smart pointers to ensure that the memory a pointer points to is automatically deallocated when the pointer is destroyed, for example if the pointer goes out of scope. When unrelated smart pointers manage the same pointer value, one of the smart pointers might attempt to deallocate memory that was already deallocated by the other smart pointer. This results in a double free vulnerability, which corrupts your program's memory management data structure.

**Fix**

Use `std::make_shared` to create a smart pointer and then use copy construction to create a related smart pointer. The underlying pointer value is managed by both smart pointers and the memory pointed to is not deallocated until all the smart pointers are destroyed.

If you do not intend to allow multiple smart pointers to manage the same pointer value, use `std::make_unique` to construct a `std::unique_ptr` smart pointer. A `std::unique_ptr` can only be moved, which relinquishes ownership of the underlying managed pointer value.

**Example — Use of an Already-Owned Pointer**

```
#include <memory>
#include <string>

struct Profile
{
    virtual ~Profile()=default;
};

struct Player : public Profile
{
    std::string name;
    std::int8_t rank;

    Player();
    Player(const std::string& name_, const std::int8_t& rank_) :
        name{ name_ }, rank{ rank_ } {}
};

void func(){

    Player * player = new Player("Richard Roll",1);
    std::shared_ptr<Player> player1(player);
    std::shared_ptr<Player> top_rank(player); //Non-compliant

}
```

In this example, the use of pointer value `player` to construct smart pointer `top_rank` in function `func` is non-compliant. `player` is already owned by smart pointer `player1`. When `player1` is destroyed, it might attempt to delete pointer value `player` which was already deleted by `top_rank`.

#### Correction — Use `std::make_shared` and Copy Construction to Create Related Smart Pointers

```
#include <memory>
#include <string>

struct Profile
{
    virtual ~Profile()=default;
};

struct Player : public Profile
{
    std::string name;
    std::int8_t rank;

    Player();
    Player(const std::string& name_, const std::int8_t& rank_) :
        name{ name_ }, rank{ rank_ } {}
};

void func2(){

    std::shared_ptr<Player> player1_shared =
        std::make_shared<Player>("Richard Roll",1);
    std::shared_ptr<Player> top_rank_shared(player1_shared); //Compliant

}
```

One possible correction is to use `std::make_shared` to declare `player1_shared`, and then use copy construction to create related smart pointer `top_rank_shared`. The underlying pointer value is not deleted until all smart pointers are destroyed.

## Check Information

**Group:** Rule 06. Memory Management (MEM)

# Version History

**Introduced in R2021a**

## See Also

`Check SEI CERT-C++ (-cert-cpp)` | `AUTOSAR C++14 Rule A20-8-1`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MEM56-CPP

# CERT C++: MEM57-CPP

Avoid using default operator new for over-aligned types

## Description

### Rule Definition

*Avoid using default operator new for over-aligned types.*

### Polyspace Implementation

The rule checker checks for **Operator new not overloaded for possibly overaligned class**.

## Examples

### Operator new not overloaded for possibly overaligned class

#### Issue

**Operator new not overloaded for possibly overaligned class** occurs when you do not adequately overload operator `new/new[]` and you use this operator to create an object with an alignment requirement specified with `alignas`. The checker raises a defect for these versions of throwing and non-throwing operator `new/new[]`.

- `void* operator new(std::size_t size)`
- `void* operator new(std::size_t size, const std::nothrow_t&)`
- `void* operator new[](std::size_t size)`
- `void* operator new[](std::size_t size, const std::nothrow_t&)`

The use of `alignas` indicates that you do not expect the default operator `new/new[]` to satisfy the alignment requirement or the object, and that the object is possibly over aligned. A type is over aligned if you use `alignas` to make the alignment requirement of the type larger than `std::max_align_t`. For instance, `foo` is over aligned in this code snippet because its alignment requirement is 32 bytes, but `std::max_align_t` has an alignment of 16 bytes in most implementations.

```
struct alignas(32) foo {
  char elems[32];
};
```

**Operator new not overloaded for possibly overaligned class** raises no defect if you do not overload the operator `new/new[]` and you use version C++17 or later of the Standard. The default operator `new/new[]` in C++17 or later supports over alignment by passing the alignment requirement as an argument of type `std::align_val_t`, for instance `void* operator new(std::size_t size, std::align_val_t alignment)`.

#### Risk

The default operator `new/new[]` allocates storage with the alignment requirement of `std::align_val_t` at most. If you do not overload the operator when you create an object with over aligned type, the resulting object may be misaligned. Accessing this object might cause illegal access errors or abnormal program terminations.

#### Fix

If you use version C++14 or earlier of the Standard, pass the alignment requirement of over aligned types to the operator `new/new[]` by overloading the operator.

#### Example - Allocated Memory Is Smaller Than Alignment Requirement of Type `foo`

```
#include <new>
#include <cstdlib>
#include <iostream>

struct alignas(64) foo {
    char elems[32];
};

foo*  func()
{
    foo*  bar = 0x0;
    try {
        bar =  new  foo ; //Noncompliant
    } catch (...) { return nullptr; }
    delete bar;
}
```

In this example, structure `foo` is declared with an alignment requirement of 32 bytes. When you use the default operator `new` to create object `bar`, the allocated memory for `bar` is smaller than the alignment requirement of type `foo` and `bar` might be misaligned.

#### Correction — Define Overloaded Operator `new` to Handle Alignment Requirement of Type `foo`

One possible correction, if you use C11 `stdlib.h` or POSIX-C `malloc.h`, is to define an overloaded operator `new` that uses `aligned_alloc()` or `posix_memalign()` or to obtain storage with the correct alignment.

```
#include <new>
#include <cstdlib>
#include <iostream>

struct alignas(64) foo {
    char elems[32];
    static void* operator new (size_t nbytes)
    {
        if (void* p =
                ::aligned_alloc(alignof(foo), nbytes)) {
            return p;
        }
        throw std::bad_alloc();
    }
    static void operator delete(void *p) {
        free(p);
    }
};

foo*  func()
{
    foo*  bar = 0x0;
    try {
        bar =  new  foo ;
    } catch (...) { return nullptr; }
    delete bar;
}
```

## Check Information

**Group:** Rule 06. Memory Management (MEM)

## Version History

**Introduced in R2019b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MEM57-CPP

# CERT C++: FIO30-C

Exclude user input from format strings

## Description

### Rule Definition

*Exclude user input from format strings.*

### Polyspace Implementation

The rule checker checks for **Tainted string format**.

### Extend Checker

A default Bug Finder analysis might not flag a **Tainted string format** issue for certain inputs that originate outside of the current analysis boundary. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

## Examples

### Tainted string format

#### Issue

**Tainted string format** detects string formatting with `printf`-style functions that contain elements from unsecure sources.

#### Risk

If you use externally controlled elements to format a string, you can cause buffer overflow or data-representation problems. An attacker can use these string formatting elements to view the contents of a stack using `%x` or write to a stack using `%n`.

#### Fix

Pass a static string to format string functions. This fix ensures that an external actor cannot control the string.

Another possible fix is to allow only the expected number of arguments. If possible, use functions that do not support the vulnerable `%n` operator in format strings.

#### Example - Get Elements from User Input

```
#include <stdio.h>
#include <unistd.h>
#define MAX 40
void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf(userstr); //Noncompliant
}
```

This example prints the input argument `userstr`. The string is unknown. If it contains elements such as `%`, `printf` can interpret `userstr` as a string format instead of a string, causing your program to crash.

#### Correction — Print as String

One possible correction is to print `userstr` explicitly as a string so that there is no ambiguity.

```
#include "stdio.h"
#include <unistd.h>
#define MAX 40

void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf("%.20s", userstr);
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO30-C

# CERT C++: FIO32-C

Do not perform operations on devices that are only appropriate for files

## Description

### Rule Definition

*Do not perform operations on devices that are only appropriate for files.*

### Polyspace Implementation

The rule checker checks for **Inappropriate I/O operation on device files**.

## Examples

### Inappropriate I/O operation on device files

#### Issue

**Inappropriate I/O operation on device files** occurs when you do not check whether a file name parameter refers to a device file before you pass it to these functions:

- `fopen()`
- `fopen_s()`
- `freopen()`
- `remove()`
- `rename()`
- `CreateFile()`
- `CreateFileA()`
- `CreateFileW()`
- `_wfopen()`
- `_wfopen_s()`

Device files are files in a file system that provide an interface to device drivers. You can use these files to interact with devices.

**Inappropriate I/O operation on device files** does not raise a defect when:

- You use `stat` or `lstat`-family functions to check the file name parameter before calling the previously listed functions.
- You use a string comparison function to compare the file name against a list of device file names.

#### Risk

Operations appropriate only for regular files but performed on device files can result in denial-of-service attacks, other security vulnerabilities, or system failures.

### Fix

Before you perform an I/O operation on a file:

- Use `stat()`, `lstat()`, or an equivalent function to check whether the file name parameter refers to a regular file.
- Use a string comparison function to compare the file name against a list of device file names.

### Example - Using `fopen()` Without Checking `file_name`

```
#include <stdio.h>
#include <string.h>

#define SIZE1024 1024

FILE* func()
{

    FILE* f;
    const char file_name[SIZE1024] = "./tmp/file";

    if ((f = fopen(file_name, "w")) == NULL) { //Noncompliant
        /*handle error */
    };
    /*operate on file */
}
```

In this example, `func()` operates on the file `file_name` without checking whether it is a regular file. If `file_name` is a device file, attempts to access it can result in a system failure.

### Correction — Check File with `lstat()` Before Calling `fopen()`

One possible correction is to use `lstat()` and the `S_ISREG` macro to check whether the file is a regular file. This solution contains a `TOCTOU` race condition that can allow an attacker to modify the file after you check it but before the call to fopen(). To prevent this vulnerability, ensure that `file_name` refers to a file in a secure folder.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>

#define SIZE1024 1024

FILE* func()
{

    FILE* f;
    const char file_name[SIZE1024] = "./tmp/file";
    struct stat orig_st;
    if ((lstat(file_name, &orig_st) != 0) ||
        (!S_ISREG(orig_st.st_mode))) {
        exit(0);
    }
    if ((f = fopen(file_name, "w")) == NULL) {
        /*handle error */
    };
    /*operate on file */
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FIO32-C

# CERT C++: FIO34-C

Distinguish between characters read from a file and EOF or WEOF

## Description

### Rule Definition

*Distinguish between characters read from a file and EOF or WEOF.*

### Polyspace Implementation

The rule checker checks for **Character value absorbed into EOF**.

## Examples

### Character value absorbed into EOF

#### Issue

**Character value absorbed into EOF** occurs when you perform a data type conversion that makes a valid character value indistinguishable from `EOF` (End-of-File). Bug Finder flags the defect in one of the following situations:

- *End-of-File*: You perform a data type conversion such as from `int` to `char` that converts a non-EOF character value into `EOF`.

  ```
  char ch = (char)getchar()
  ```

  You then compare the result with EOF.

  ```
  if((int)ch == EOF)
  ```

  The conversion can be explicit or implicit.
- *Wide End-of-File*: You perform a data type conversion that can convert a non-WEOF wide character value into WEOF, and then compare the result with WEOF.

#### Risk

The data type `char` cannot hold the value `EOF` that indicates the end of a file. Functions such as `getchar` have return type `int` to accommodate `EOF`. If you convert from `int` to `char`, the values `UCHAR_MAX` (a valid character value) and `EOF` get converted to the same value -1 and become indistinguishable from each other. When you compare the result of this conversion with `EOF`, the comparison can lead to false detection of `EOF`. This rationale also applies to wide character values and `WEOF`.

#### Fix

Perform the comparison with `EOF` or `WEOF` before conversion.

#### Example - Return Value of `getchar` Converted to `char`

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define fatal_error() abort()

char func(void)
{
    char ch;
    ch = getchar();
    if (EOF == (int)ch) { //Noncompliant
        fatal_error();
    }
    return ch;
}
```

In this example, the return value of `getchar` is implicitly converted to `char`. If `getchar` returns `UCHAR_MAX`, it is converted to -1, which is indistinguishable from EOF. When you compare with EOF later, it can lead to a false positive.

#### Correction — Perform Comparison with EOF Before Conversion

One possible correction is to first perform the comparison with EOF, and then convert from `int` to `char`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

char func(void)
{
    int i;
    i = getchar();
    if (EOF == i) {
        fatal_error();
    }
    else {
        return (char)i;
    }
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

FIO34-C

# CERT C++: FIO37-C

Do not assume that fgets() or fgetws() returns a nonempty string when successful

## Description

### Rule Definition

*Do not assume that fgets() or fgetws() returns a nonempty string when successful.*

### Polyspace Implementation

The rule checker checks for **Use of indeterminate string**.

## Examples

### Use of indeterminate string

#### Issue

**Use of indeterminate string** occurs when you do not check the validity of the buffer returned from `fgets`-family functions. The checker raises a defect when such a buffer is used as:

- An argument in standard functions that print or manipulate strings or wide strings.
- A return value.
- An argument in external functions with parameter type `const char *` or `const wchar_t *`.

#### Risk

If an `fgets`-family function fails, the content of its output buffer is indeterminate. Use of such a buffer has undefined behavior and can result in a program that stops working or other security vulnerabilities.

#### Fix

Reset the output buffer of an `fgets`-family function to a known string value when the function fails.

#### Example - Output of `fgets()` Passed to External Function

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func(void) {
    char buf[SIZE20];

    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* 'buf' may contain an indeterminate string.  */
        ;
    }
    /* 'buf passed to external function */
    display_text(buf); //Noncompliant
}
```

In this example, the output `buf` is passed to the external function `display_text()`, but its value is not reset if `fgets()` fails.

#### Correction — Reset `fgets()` Output on Failure

If `fgets()` fails, reset `buf` to a known value before you pass it to an external function.

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func1(void) {
    char buf[SIZE20];
    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* value of 'buf' reset after fgets() failure. */
        buf[0] = '\0';
    }
    /* 'buf' passed to external function */
    display_text(buf);
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO37-C

# CERT C++: FIO38-C

Do not copy a FILE object

## Description

### Rule Definition

*Do not copy a FILE object.*

### Polyspace Implementation

The rule checker checks for **Misuse of a FILE object**.

## Examples

### Misuse of a FILE object

#### Issue

**Misuse of a FILE object** occurs when:

- You dereference a pointer to a FILE object, including indirect dereference by using `memcmp()`.
- You modify an entire FILE object or one of its components through its pointer.
- You take the address of FILE object that was not returned from a call to an `fopen`-family function. No defect is raised if a macro defines the pointer as the address of a built-in FILE object, such as `#define ptr (&__stdout)`.

#### Risk

In some implementations, the address of the pointer to a FILE object used to control a stream is significant. A pointer to a copy of a FILE object is interpreted differently than a pointer to the original object, and can potentially result in operations on the wrong stream. Therefore, the use of a copy of a FILE object can cause the software to stop responding, which an attacker might exploit in denial-of-service attacks.

#### Fix

Do not make a copy of a FILE object. Do not use the address of a FILE object that was not returned from a successful call to an `fopen`-family function.

#### Example - Copy of FILE Object Used in `fputs()`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

void fatal_error(void);

int func(void)
{
    /*'stdout' dereferenced and contents
        copied to 'my_stdout'. */
    FILE my_stdout = *stdout;  //Noncompliant

    /* Address of 'my_stdout' may not point to correct stream. */
    if (fputs("Hello, World!\n", &my_stdout) == EOF)  //Noncompliant
    {
        /* Handler error */
        fatal_error();
    }
    return 0;
}
```

In this example, FILE object `stdout` is dereferenced and its contents are copied to `my_stdout`. The contents of `stdout` might not be significant. `fputs()` is then called with the address of `my_stdout` as an argument. Because no call to `fopen()` or a similar function was made, the address of `my_stdout` might not point to the correct stream.

**Correction — Copy the FILE Object Pointer**

Declare `my_stdout` to point to the same address as `stdout` to ensure that you write to the correct stream when you call `fputs()`.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

void fatal_error(void);

int func(void)
{
    /* 'my_stdout' and 'stdout' point to the same object. */
    FILE *my_stdout = stdout;
    if (fputs("Hello, World!\n", my_stdout) == EOF)
    {
        /* Handler error */
        fatal_error();
    }
    return 0;
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**
"Check for and Review Coding Standard Violations"

**External Websites**
FIO38-C

# CERT C++: FIO39-C

Do not alternately input and output from a stream without an intervening flush or positioning call

## Description

### Rule Definition

*Do not alternately input and output from a stream without an intervening flush or positioning call.*

### Polyspace Implementation

The rule checker checks for **Alternating input and output from a stream without flush or positioning call**.

## Examples

### Alternating input and output from a stream without flush or positioning call

#### Issue

**Alternating input and output from a stream without flush or positioning call** occurs when:

- You do not perform a flush or function positioning call between an output operation and a following input operation on a file stream in update mode.
- You do not perform a function positioning call between an input operation and a following output operation on a file stream in update mode.

#### Risk

Alternating input and output operations on a stream without an intervening flush or positioning call is undefined behavior.

#### Fix

Call `fflush()` or a file positioning function such as `fseek()` or `fsetpos()` between output and input operations on an update stream.

Call a file positioning function between input and output operations on an update stream.

#### Example - Read After Write Without Intervening Flush

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";

void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;
    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Read operation after write without
    intervening flush. */
    if (fread(data, 1, SIZE20, file) < SIZE20) //Noncompliant
      {
          (void)fclose(file);
          /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

In this example, the file `demo.txt` is opened for reading and appending. After the call to `fwrite()`, a call to `fread()` without an intervening flush operation is undefined behavior.

**Correction — Call `fflush()` Before the Read Operation**

After writing data to the file, before calling `fread()`, perform a flush call.

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";


void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
```

```
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Buffer flush after write and before read */
    if (fflush(file) != 0)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO39-C

# CERT C++: FIO40-C

Reset strings on fgets() or fgetws() failure

## Description

### Rule Definition

*Reset strings on fgets() or fgetws() failure.*

### Polyspace Implementation

The rule checker checks for **Use of indeterminate string**.

## Examples

### Use of indeterminate string

#### Issue

**Use of indeterminate string** occurs when you do not check if a write operation using an `fgets`-family function such as:

```
char * fgets(char* buf, int n, FILE *stream)
```

succeeded and the buffer written has valid content, or you do not reset the buffer on failure. You then perform an operation that assumes a buffer with valid content. For instance, if the buffer with possibly indeterminate content is `buf` (as shown above), the checker raises a defect if:

- You pass `buf` as argument to standard functions that print or manipulate strings or wide strings.
- You return `buf` from a function.
- You pass `buf` as argument to external functions with parameter type `const char *` or `const wchar_t *`.
- You read `buf` as `buf[index]` or `*(buf + offset)`, where `index` or `offset` is a numerical value representing the distance from the beginning of the buffer.

#### Risk

If an `fgets`-family function fails, the content of its output buffer is indeterminate. Use of such a buffer has undefined behavior and can result in a program that stops working or other security vulnerabilities.

#### Fix

Reset the output buffer of an `fgets`-family function to a known string value when the function fails.

#### Example - Output of `fgets()` Passed to External Function

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>
```

```
#define SIZE20 20

extern void display_text(const char *txt);

void func(void) {
    char buf[SIZE20];

    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* 'buf' may contain an indeterminate string.  */
        ;
    }
    /* 'buf passed to external function */
    display_text(buf); //Noncompliant
}
```

In this example, the output `buf` is passed to the external function `display_text()`, but its value is not reset if `fgets()` fails.

#### Correction — Reset `fgets()` Output on Failure

If `fgets()` fails, reset `buf` to a known value before you pass it to an external function.

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func1(void) {
    char buf[SIZE20];
    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* value of 'buf' reset after fgets() failure. */
        buf[0] = '\0';
    }
    /* 'buf' passed to external function */
    display_text(buf);
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FIO40-C

# CERT C++: FIO41-C

Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects

## Description

### Rule Definition

*Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects.*

### Polyspace Implementation

The rule checker checks for **Stream argument with possibly unintended side effects**.

## Examples

### Stream argument with possibly unintended side effects

#### Issue

**Stream argument with possibly unintended side effects** occurs when you call `getc()`, `putc()`, `getwc()`, or `putwc()` with a stream argument that has side effects.

**Stream argument with possibly unintended side effects** considers the following as stream side effects:

- Any assignment of a variable of a stream, such as `FILE *`, or any assignment of a variable of a deeper stream type, such as an array of `FILE *`.
- Any call to a function that manipulates a stream or a deeper stream type.

The number of defects raised corresponds to the number of side effects detected. When a stream argument is evaluated multiple times in a function implemented as a macro, a defect is raised for each evaluation that has a side effect.

A defect is also raised on functions that are not implemented as macros but that can be implemented as macros on another operating system.

#### Risk

If the function is implemented as an unsafe macro, the stream argument can be evaluated more than once, and the stream side effect happens multiple times. For instance, a stream argument calling `fopen()` might open the same file multiple times, which is unspecified behavior.

#### Fix

To ensure that the side effect of a stream happens only once, use a separate statement for the stream argument.

#### Example - Stream Argument of `getc()` Has Side Effect `fopen()`

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#define fatal_error() abort()

const char* myfile = "my_file.log";

void func(void)
{
    int c;
    FILE* fptr;
    /* getc() has stream argument fptr with
    * 2 side effects: call to fopen(), and assignment
    * of fptr
    */
     c = getc(fptr = fopen(myfile, "r"));//Noncompliant
    if (c == EOF) {
        /* Handle error */
        (void)fclose(fptr);
        fatal_error();
    }
    if (fclose(fptr) == EOF) {
        /* Handle error */
        fatal_error();
    }
}

void main(void)
{
    func();

}
```

In this example, `getc()` is called with stream argument `fptr`. The stream argument has two side effects: the call to `fopen()` and the assignment of `fptr`. If `getc()` is implemented as an unsafe macro, the side effects happen multiple times.

#### Correction — Use Separate Statement for `fopen()`

One possible correction is to use a separate statement for `fopen()`. The call to `fopen()` and the assignment of `fptr` happen in this statement so there are no side effects when you pass `fptr` to `getc()`.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

#define fatal_error() abort()


const char* myfile = "my_file.log";

void func(void)
{
    int c;
    FILE* fptr;

    /* Separate statement for fopen()
    * before call to getc()
    */
    fptr = fopen(myfile, "r");
    if (fptr == NULL) {
        /* Handle error */
        fatal_error();
    }
    c = getc(fptr);
    if (c == EOF) {
        /* Handle error */
        (void)fclose(fptr);
        fatal_error();
    }
    if (fclose(fptr) == EOF) {
        /* Handle error */
        fatal_error();
    }
}

void main(void)
{
    func();

}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FIO41-C

# CERT C++: FIO42-C

Close files when they are no longer needed

## Description

### Rule Definition

*Close files when they are no longer needed.*

### Polyspace Implementation

The rule checker checks for **Resource leak**.

## Examples

### Resource leak

#### Issue

**Resource leak** occurs when you open a file stream by using a `FILE` pointer but do not close it before:

- The end of the pointer’s scope.
- Assigning the pointer to another stream.

#### Risk

If you do not release file handles explicitly as soon as possible, a failure can occur due to exhaustion of resources.

#### Fix

Close a `FILE` pointer before the end of its scope, or before you assign the pointer to another stream.

#### Example - FILE Pointer Not Released Before End of Scope

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" ); //Noncompliant
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`.

#### Correction — Release FILE Pointer

One possible correction is to explicitly dissociate `fp1` from the file stream of `data1.txt`.

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );
    fclose(fp1);

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FIO42-C

# CERT C++: FIO44-C

Only use values for fsetpos() that are returned from fgetpos()

## Description

### Rule Definition

*Only use values for fsetpos() that are returned from fgetpos().*

### Polyspace Implementation

The rule checker checks for **Invalid file position**.

## Examples

### Invalid file position

#### Issue

**Invalid file position** occurs when the file position argument of `fsetpos()` uses a value that is not obtained from `fgetpos()`.

#### Risk

The function `fgetpos(FILE *stream, fpos_t *pos)` gets the current file position of the stream. When you use any other value as the file position argument of `fsetpos(FILE *stream, const fpos_t *pos)`, you might access an unintended location in the stream.

#### Fix

Use the value returned from a successful call to `fgetpos()` as the file position argument of `fsetpos()`.

#### Example - `memset()` Sets File Position Argument

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset' */
    (void)memset(&offset, 0, sizeof(offset));

    /* Read data from file */

    /* Return to the initial position. offset was not
    returned from a call to fgetpos()    */
    if (fsetpos(file, &offset) != 0) //Noncompliant
    {
        /* Handle error */
    }
    return file;
}
```

In this example, `fsetpos()` uses `offset` as its file position argument. However, the value of `offset` is set by `memset()`. The preceding code might access the wrong location in the stream.

#### Correction — Use a File Position Returned From `fgetpos()`

Call `fgetpos()`, and if it returns successfully, use the position argument in your call to `fsetpos()`.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

FILE *func(FILE *file)
{
    fpos_t offset;
    if (file == NULL)
    {
        /* Handle error */
    }
    /* Store initial position in variable 'offset'
    using fgetpos() */
    if (fgetpos(file, &offset) != 0)
    {
        /* Handle error */
    }

    /* Read data from file */

    /* Back to the initial position */
    if (fsetpos(file, &offset) != 0)
    {
        /* Handle error */
    }
    return file;
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO44-C

# CERT C++: FIO45-C

Avoid TOCTOU race conditions while accessing files

## Description

### Rule Definition

*Avoid TOCTOU race conditions while accessing files.*

### Polyspace Implementation

The rule checker checks for **File access between time of check and use (TOCTOU)**.

## Examples

### File access between time of check and use (TOCTOU)

#### Issue

**File access between time of check and use (TOCTOU)** detects race condition issues between checking the existence of a file or folder, and using a file or folder.

#### Risk

An attacker can access and manipulate your file between your check for the file and your use of a file. Symbolic links are particularly risky because an attacker can change where your symbolic link points.

#### Fix

Before using a file, do not check its status. Instead, use the file and check the results afterward.

#### Example - Check File Before Using

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    if (access(log_path, W_OK)==0) {
        FILE* f = fopen(log_path, "w"); //Noncompliant
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

In this example, before opening and using the file, the function checks if the file exists. However, an attacker can change the file between the first and second lines of the function.

#### Correction — Open Then Check

One possible correction is to open the file, and then check the existence and contents afterward.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern void print_tofile(FILE* f);

void toctou(char * log_path) {
    int fd = open(log_path, O_WRONLY);
    if (fd!=-1) {
        FILE *f = fdopen(fd, "w");
        if (f) {
            print_tofile(f);
            fclose(f);
        }
    }
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO45-C

# CERT C++: FIO46-C

Do not access a closed file

## Description

### Rule Definition

*Do not access a closed file.*

### Polyspace Implementation

The rule checker checks for **Use of previously closed resource**.

## Examples

### Use of previously closed resource

#### Issue

**Use of previously closed resource** occurs when a function operates on a stream that you closed earlier in your code.

#### Risk

The standard states that the value of a `FILE*` pointer is indeterminate after you close the stream associated with it. Operations using the `FILE*` pointer can produce unintended results.

#### Fix

One possible fix is to close the stream only at the end of operations. Another fix is to reopen the stream before using it again.

#### Example - Use of `FILE*` Pointer After Closing Stream

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fclose(fp);
        fprintf(fp,"text"); //Noncompliant
    }
}
```

In this example, `fclose` closes the stream associated with `fp`. When you use `fprintf` on `fp` after `fclose`, the **Use of previously closed resource** defect appears.

#### Correction — Close Stream After All Operations

One possible correction is to reverse the order of the `fprintf` and `fclose` operations.

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fprintf(fp,"text");
        fclose(fp);
    }
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C++ (-cert-cpp))

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FIO46-C

# CERT C++: FIO47-C

Use valid format strings

## Description

### Rule Definition

*Use valid format strings.*

### Polyspace Implementation

The rule checker checks for **Format string specifiers and arguments mismatch**.

## Examples

### Format string specifiers and arguments mismatch

#### Issue

**Format string specifiers and arguments mismatch** occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

#### Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

#### Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
const char *message = "License not available";
int err_number = -4;
printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Printing a Float

```
#include <stdio.h>

void string_format(void) {
    unsigned long fst = 1;

    printf("%d\n", fst); //Noncompliant
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

**Correction — Use an Unsigned Long Format Specifier**

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

**Correction — Use an Integer Argument**

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FIO47-C

# CERT C++: FIO50-CPP

Do not alternately input and output from a file stream without an intervening positioning call

## Description

### Rule Definition

*Do not alternately input and output from a file stream without an intervening positioning call.*

### Polyspace Implementation

The rule checker checks for **Alternating input and output from a stream without flush or positioning call**.

## Examples

### Alternating input and output from a stream without flush or positioning call

#### Issue

**Alternating input and output from a stream without flush or positioning call** occurs when:

- You do not perform a flush or function positioning call between an output operation and a following input operation on a file stream in update mode.
- You do not perform a function positioning call between an input operation and a following output operation on a file stream in update mode.

#### Risk

Alternating input and output operations on a stream without an intervening flush or positioning call is undefined behavior.

#### Fix

Call `fflush()` or a file positioning function such as `fseek()` or `fsetpos()` between output and input operations on an update stream.

Call a file positioning function between input and output operations on an update stream.

#### Example - Read After Write Without Intervening Flush

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";

void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;
    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Read operation after write without
    intervening flush. */
    if (fread(data, 1, SIZE20, file) < SIZE20) //Noncompliant
      {
          (void)fclose(file);
          /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

In this example, the file `demo.txt` is opened for reading and appending. After the call to `fwrite()`, a call to `fread()` without an intervening flush operation is undefined behavior.

### Correction — Call `fflush()` Before the Read Operation

After writing data to the file, before calling `fread()`, perform a flush call.

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";


void func()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
```

```
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Buffer flush after write and before read */
    if (fflush(file) != 0)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    if (fread(data, 1, SIZE20, file) < SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FIO50-CPP

# CERT C++: FIO51-CPP

Close files when they are no longer needed

## Description

### Rule Definition

*Close files when they are no longer needed.*

### Polyspace Implementation

The rule checker checks for **Resource leak**.

## Examples

### Resource leak

#### Issue

**Resource leak** occurs when you open a file stream by using a `FILE` pointer but do not close it before:

- The end of the pointer’s scope.
- Assigning the pointer to another stream.

#### Risk

If you do not release file handles explicitly as soon as possible, a failure can occur due to exhaustion of resources.

#### Fix

Close a `FILE` pointer before the end of its scope, or before you assign the pointer to another stream.

#### Example - FILE Pointer Not Released Before End of Scope

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" ); //Noncompliant
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`.

#### Correction — Release FILE Pointer

One possible correction is to explicitly dissociate `fp1` from the file stream of `data1.txt`.

```
#include <stdio.h>

void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );
    fclose(fp1);

    fp1 = fopen ( "data2.txt", "w" );
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}
```

## Check Information

**Group:** 07. Input Output (FIO)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FIO51-CPP

# CERT C++: ERR30-C

Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure

## Description

### Rule Definition

*Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Misuse of errno**.
- **Errno not reset**.

## Examples

### Misuse of errno

#### Issue

**Misuse of errno** occurs when you check `errno` for error conditions in situations where checking `errno` does not guarantee the absence of errors. In some cases, checking `errno` can lead to false positives.

For instance, you check `errno` following calls to the functions:

- `fopen`: If you follow the ISO Standard, the function might not set `errno` on errors.
- `atof`: If you follow the ISO Standard, the function does not set `errno`.
- `signal`: The `errno` value indicates an error only if the function returns the `SIG_ERR` error indicator.

#### Risk

The ISO C Standard does not enforce that these functions set `errno` on errors. Whether the functions set `errno` or not is implementation-dependent.

To detect errors, if you check `errno` alone, the validity of this check also becomes implementation-dependent.

In some cases, the `errno` value indicates an error only if the function returns a specific error indicator. If you check `errno` before checking the function return value, you can see false positives.

#### Fix

For information on how to detect errors, see the documentation for that specific function.

Typically, the functions return an out-of-band error indicator to indicate errors. For instance:

- fopen returns a null pointer if an error occurs.
- signal returns the SIG_ERR error indicator and sets errno to a positive value. Check errno only after you have checked the function return value.

**Example - Incorrectly Checking for errno After fopen Call**

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    errno = 0;
    fileptr = fopen(temp_filename, "w+b");
    if (errno != 0) { //Noncompliant
        if (fileptr != NULL) {
            (void)fclose(fileptr);
        }
        /* Handle error */
        fatal_error();
    }
    return fileptr;
}
```

In this example, errno is the first variable that is checked after a call to fopen. You might expect that fopen changes errno to a nonzero value if an error occurs. If you run this code with an implementation of fopen that does not set errno on errors, you might miss an error condition. In this situation, fopen can return a null pointer that escapes detection.

**Correction — Check Return Value of fopen After Call**

One possible correction is to only check the return value of fopen for a null pointer.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define fatal_error() abort()

const char *temp_filename = "/tmp/demo.txt";

FILE *func()
{
    FILE *fileptr;
    fileptr = fopen(temp_filename, "w+b");
    if (fileptr == NULL) {
        fatal_error();
    }
    return fileptr;
}
```

### Errno not reset

#### Issue

**Errno not reset** occurs when you do not reset `errno` before calling a function that sets `errno` to indicate error conditions. However, you check `errno` for those error conditions after the function call.

#### Risk

The `errno` is not clean and can contain values from a previous call. Checking `errno` for errors can give the false impression that an error occurred.

`errno` is set to zero at program startup but subsequently, `errno` is not reset by a C standard library function. You must explicitly set `errno` to zero when required.

#### Fix

Before calling a function that sets `errno` to indicate error conditions, reset `errno` to zero explicitly.

#### Example - `errno` Not Reset Before Call to `strtod`

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    f1 = strtod (s1, NULL);
    if (0 == errno) {  //Noncompliant
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
                  {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

In this example, `errno` is not reset to 0 before the first call to `strtod`. Checking `errno` for 0 later can lead to a false positive.

#### Correction — Reset `errno` Before Call

One possible correction is to reset `errno` to 0 before calling `strtod`.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>
```

```
#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    errno = 0;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
              {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

### Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

ERR30-C

# CERT C++: ERR32-C

Do not rely on indeterminate values of errno

## Description

### Rule Definition

*Do not rely on indeterminate values of errno.*

### Polyspace Implementation

The rule checker checks for **Misuse of errno in a signal handler**.

## Examples

### Misuse of errno in a signal handler

#### Issue

**Misuse of errno in a signal handler** occurs when you call one of these functions in a signal handler:

- `signal`: You call the `signal` function in a signal handler and then read the value of `errno`.

  For instance, the signal handler function `handler` calls `signal` and then calls `perror`, which reads `errno`.

  ```
  void handler(int signum) {
    pfv old_handler = signal(signum, SIG_DFL);
    if (old_handler == SIG_ERR) {
      perror("SIGINT handler");
    }
  }
  ```
- `errno`-setting POSIX function: You call an `errno`-setting POSIX function in a signal handler but do not restore `errno` when returning from the signal handler.

  For instance, the signal handler function `handler` calls `waitpid`, which changes `errno`, but does not restore `errno` before returning.

  ```
  void handler(int signum) {
    int rc = waitpid(-1, NULL, WNOHANG);
    if (ECHILD != errno) {
    }
  }
  ```

#### Risk

In each case that the checker flags, you risk relying on an indeterminate value of `errno`.

- `signal`: If the call to `signal` in a signal handler fails, the value of `errno` is indeterminate (see C11 Standard, Sec. 7.14.1.1). If you rely on a specific value of `errno`, you can see unexpected results.

- errno-setting POSIX function: An errno-setting function sets errno on failure. If you read errno after a signal handler is called and the signal handler itself calls an errno-setting function, you can see unexpected results.

**Fix**

Avoid situations where you risk relying on an indeterminate value of errno.

- signal: After calling the signal function in a signal handler, do not read errno or use a function that reads errno.
- errno-setting POSIX function: Before calling an errno-setting function in a signal handler, save errno to a temporary variable. Restore errno from this variable before returning from the signal handler.

**Example - Reading errno After signal Call in Signal Handler**

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#define fatal_error() abort()

void handler(int signum) {
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        perror("SIGINT handler");
    }
} //Noncompliant

int func(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
        fatal_error();
    }
    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
        fatal_error();
    }
    return 0;
}
```

In this example, the function handler is called to handle the SIGINT signal. In the body of handler, the signal function is called. Following this call, the value of errno is indeterminate. The checker raises a defect when the perror function is called because perror relies on the value of errno.

**Correction — Avoid Reading errno After signal Call**

One possible correction is to not read errno after calling the signal function in a signal handler. The corrected code here calls the abort function via the fatal_error macro instead of the perror function.

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#define fatal_error() abort()
```

```
void handler(int signum) {
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        fatal_error();
    }
}

int func(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
        fatal_error();
    }
    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
        fatal_error();
    }
    return 0;
}
```

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR32-C

# CERT C++: ERR33-C

Detect and handle standard library errors

## Description

### Rule Definition

*Detect and handle standard library errors.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Errno not checked**.
- **Returned value of a sensitive standard function not checked**.
- **Unprotected dynamic memory allocation**.

## Examples

### Errno not checked

#### Issue

**Errno not checked** occurs when you call a function that sets errno to indicate error conditions, but do not check errno after the call. For these functions, checking errno is the only reliable way to determine if an error occurred.

Functions that set errno on errors include:

- fgetwc, strtol, and wcstol.

  For a comprehensive list of functions, see documentation about errno.
- POSIX errno-setting functions such as encrypt and setkey.

#### Risk

To see if the function call completed without errors, check errno for error values.

The return values of these errno-setting functions do not indicate errors. The return value can be one of the following:

- void
- Even if an error occurs, the return value can be the same as the value from a successful call. Such return values are called in-band error indicators.

You can determine if an error occurred only by checking errno.

For instance, strtol converts a string to a long integer and returns the integer. If the result of conversion overflows, the function returns LONG_MAX and sets errno to ERANGE. However, the function can also return LONG_MAX from a successful conversion. Only by checking errno can you distinguish between an error and a successful conversion.

#### Fix

Before calling the function, set `errno` to zero.

After the function call, to see if an error occurred, compare `errno` to zero. Alternatively, compare `errno` to known error indicator values. For instance, `strtol` sets `errno` to `ERANGE` to indicate errors.

The error message in the Polyspace result shows the error indicator value that you can compare to.

#### Example - `errno` Not Checked After Call to `strtol`

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    long val = strtol(str, &endptr, base); //Noncompliant
    printf("Return value of strtol() = %ld\n", val);
}
```

You are using the return value of `strtol` without checking `errno`.

#### Correction — Check `errno` After Call

Before calling `strtol`, set `errno` to zero . After a call to `strtol`, check the return value for `LONG_MIN` or `LONG_MAX` and `errno` for `ERANGE`.

```
#include<stdlib.h>
#include<stdio.h>
#include<errno.h>
#include<limits.h>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    errno = 0;
    long val = strtol(str, &endptr, base);
    if((val == LONG_MIN || val == LONG_MAX) && errno == ERANGE) {
         printf("strtol error");
         exit(EXIT_FAILURE);
    }
    printf("Return value of strtol() = %ld\n", val);
}
```

### Returned value of a sensitive standard function not checked

#### Issue

**Returned value of a sensitive standard function not checked** occurs when you call sensitive standard functions, but you:

- Ignore the return value.
- Use an output or a return value without testing the validity of the return value.

For this defect, two type of functions are considered: ***sensitive*** and ***critical sensitive***.

A ***sensitive*** function is a standard function that can encounter:

- Exhausted system resources (for example, when allocating resources)
- Changed privileges or permissions
- Tainted sources when reading, writing, or converting data from external sources
- Unsupported features despite an existing API

A ***critical sensitive*** function is a sensitive function that performs one of these critical or vulnerable tasks:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `thrd_create`)
- Lock or unlock memory segments (for example, `mlock`)

#### Risk

If you do not check the return value of functions that perform sensitive or critical sensitive tasks, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

#### Fix

Before continuing with the program, test the return value of *critical sensitive* functions.

For *sensitive functions*, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

#### Example - Sensitive Function Return Ignored

```
#include<cstdlib>
#include<cstdio>
#include <wchar.h>
#include <locale.h>
void initialize(size_t n, size_t* size, wchar_t *wcs, const char *utf8) {

    scanf("%d",&n); //Noncompliant
    setlocale (LC_CTYPE, "en_US.UTF-8");   //Noncompliant
    *size = mbstowcs (wcs, utf8, n);
}
```

This example shows a call to the sensitive function `scanf()`. The return value of `scanf()` is ignored, causing a defect. Similarly, the pointer returned by `setlocale` is not checked. When `setlocal` returns a `NULL` pointer, the call to `mbstowcs` might fail or produce unexpected results. Polyspace flags these calls to sensitive functions when their returns are not checked.

#### Correction — Cast Function to `(void)`

One possible correction is to cast the functions to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of these sensitive functions.

```
#include<cstdlib>
#include<cstdio>
#include <wchar.h>
#include <locale.h>
void initialize(size_t n, size_t* size, wchar_t *wcs, const char *utf8) {

    (void)scanf("%d",&n); //Compliant
    (void)setlocale (LC_CTYPE, "en_US.UTF-8");   //Compliant
    *size = mbstowcs (wcs, utf8, n);
}
```

#### Correction — Test Return Value

One possible correction is to test the return value of `scanf` and `setlocale` to check for errors.

```
#include<cstdlib>
#include<cstdio>
#include <wchar.h>
#include <locale.h>
void initialize(size_t n, size_t* size, wchar_t *wcs, const char *utf8) {

    int flag = scanf("%d",&n);
    if(flag>0){ //Compliant
        // action
    }
    char* status = setlocale (LC_CTYPE, "en_US.UTF-8");
    if(status!=NULL){//Compliant
        *size = mbstowcs (wcs, utf8, n);
    }

}
```

#### Example — Unchecked Dynamic Memory Allocation

```
#include <stddef.h>
#include <stdlib.h>

void unchecked_memory_allocation(void) {
  int * p = (int*)calloc(5, sizeof(int));// C-style allocation
  *p = 2;                                  //Noncompliant
  //...
  delete[] p;
}
```

In this example, memory is dynamically allocated for the pointer `*p`. The pointer is then used without checking the output of the dynamic memory allocation operation. Polyspace raises this defect when pointers are used after an unchecked dynamic memory allocation operation.

#### Correction — Check Output of Dynamic Memory Allocation

The correction for this defect is to check the return value of the operation `new` to verify that the function performed as expected.

```
#include <stddef.h>
#include <stdlib.h>

void checked_memory_allocation(void) {
    int * p = new int[5];
    if(p==NULL){// Check output of new
        //Handle memory allocation error
    }else{
        *p = 2; //Compliant
        //...
        delete[] p;
    }

}
```

### Unprotected dynamic memory allocation

#### Issue

**Unprotected dynamic memory allocation** occurs when you do not check after dynamic memory allocation whether the memory allocation succeeded.

#### Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

#### Fix

Check the return value of `malloc`, `calloc`, or `realloc` for NULL before accessing the allocated memory location.

```
#DEFINE SIZE 8;

int *ptr = malloc(SIZE * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

#### Example - Unprotected Dynamic Memory Allocation Error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));
  *p = 2;  //Noncompliant
  /* Defect: p is not checked for NULL value */
  free(p);
} /*Defect: p is not checked for NULL before deallocating*/
```

If the memory allocation fails, the function `calloc` returns `NULL` to `p`. Before accessing the memory through `p` or freeing `p`, the code does not check whether `p` is `NULL`. These operations might result in memory leaks.

#### Correction — Check for NULL Value

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));
   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;
   free(p);
 }
```

### Pointer overwritten during reallocation

#### Issue

**Pointer overwritten during reallocation** occurs when you overwrite the original pointer by the return value of `realloc()`. For instance:

```
p = realloc(p,SIZE);
```

#### Risk

The function `realloc()` returns a `NULL` value when memory allocation fails. In the preceding code, because you overwrite `p` by the return of `realloc()`, it becomes `NULL` when the reallocation operation fails. You lose the connection between the original memory block and `p`, resulting in a memory leak.

#### Fix

When reallocating pointers, preserve the original pointer. For instance, you might use a temporary variable to store the reallocated memory.

#### Example — Avoid Overwriting Original Pointer When Reallocating Memory

```
#include <stdlib.h>

void foo (int* ptrI, size_t new_size)
{

  if (new_size == 0) {
    /* Handle error */
    return;
  }

  ptrI = (int*)realloc (ptrI, new_size);   //Noncompliant

  if (ptrI == NULL) {
    /* Handle error */
    return;
  }
}
```

Overwriting the pointer ptrI by the pointer returned by realloc destroys the association between ptrI and the original memory block. If realloc fails, such overwriting might cause a memory leak and data loss.

#### Correction — Store Reallocated Memory in Temporary Variable

When reallocating a pointer, use a temporary variable to hold the reallocated memory. Before assigning the temporary variable to ptrI, check it for NULL value to avoid memory leaks and data loss.

```
#include <stdlib.h>

void foo (int* ptrI, size_t new_size)
{
int* temp;
  if (new_size == 0) {
    /* Handle error */
    return;
  }

  temp = (int*)realloc (ptrI, new_size);   //Compliant

  if (temp == NULL) {
    /* Handle error */
    return;
  }else{
      ptrI = temp;
  }
}
```

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C++ (-cert-cpp))

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR33-C

# CERT C++: ERR34-C

Detect errors when converting a string to a number

## Description

### Rule Definition

*Detect errors when converting a string to a number.*

### Polyspace Implementation

The rule checker checks for **Unsafe conversion from string to numerical value**.

## Examples

### Unsafe conversion from string to numerical value

#### Issue

**Unsafe conversion from string to numerical value** detects conversions from strings to integer or floating-point values. If your conversion method does not include robust error handling, a defect is raised.

#### Risk

Converting a string to numerical value can cause data loss or misinterpretation. Without validation of the conversion or error handling, your program continues with invalid values.

#### Fix

- Add additional checks to validate the numerical value.
- Use a more robust string-to-numeric conversion function such as `strtol`, `strtoll`, `strtoul`, or `strtoull`.

#### Example - Conversion With `atoi`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char* argv1)
{
    int s = 0;
    if (demo_check_string_not_empty(argv1))
    {
        s = atoi(argv1); //Noncompliant
    }
    return s;
}
```

In this example, `argv1` is converted to an integer with `atoi`. `atoi` does not provide errors for an invalid integer string. The conversion can fail unexpectedly.

**Correction — Use `strtol` instead**

One possible correction is to use `strtol` to validate the input string and the converted integer.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char *argv1)
{
    char *c_str = argv1;
    char *end;
    long sl;

    if (demo_check_string_not_empty(c_str))
    {
        errno = 0; /* set errno for error check */
        sl = strtol(c_str, &end, 10);
        if (end == c_str)
        {
            (void)fprintf(stderr, "%s: not a decimal number\n", c_str);
        }
        else if ('\0' != *end)
        {
            (void)fprintf(stderr, "%s: extra characters: %s\n", c_str, end);
        }
        else if ((LONG_MIN == sl || LONG_MAX == sl) && ERANGE == errno)
        {
            (void)fprintf(stderr, "%s out of range of type long\n", c_str);
        }
        else if (sl > INT_MAX)
        {
            (void)fprintf(stderr, "%ld greater than INT_MAX\n", sl);
        }
        else if (sl < INT_MIN)
        {
            (void)fprintf(stderr, "%ld less than INT_MIN\n", sl);
        }
        else
        {
            return (int)sl;
        }
    }
    return 0;
}
```

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

ERR34-C

# CERT C++: ERR50-CPP

Do not abruptly terminate the program

## Description

### Rule Definition

*Do not abruptly terminate the program.*

### Polyspace Implementation

The rule checker checks for **Implicit call to terminate() function**.

## Examples

### Implicit call to `terminate()` function

#### Issue

The checker flags situations that might result in calling the function `std::terminate()` implicitly. These situations might include:

- An exception remains unhandled. For instance:
  - While handling an exception, it escapes through another function that raises an unhandled exception. For instance, a catch statement or exception handler invokes another function that raises an unhandled exception.
  - An empty `throw` statement raises an unhandled exception again.
- A class destructor raises an exception.
- A termination handler that is passed to `std::atexit` raises an unhandled exception.

#### Risk

Depending on the hardware and software that you use, calling `terminate()` implicitly might result in a call to `std::abort()`, which aborts program execution without deleting the variables in the stack. Such an abnormal termination results in memory leaks and security vulnerabilities.

#### Fix

To avoid implicit calls to `terminate()`:

- Avoid unhandled exceptions. For instance, execute the operations of `main()` or task main functions in a `try-catch` block. In the catch blocks:
  - Handle exceptions of type `std::exception` explicitly in appropriate catch blocks.
  - Handle the base class of exceptions arising from third-party libraries.
  - Handle unexpected exceptions in a `catch(...)` block.
- Declare destructors as `noexcept` and handle the exceptions in destructors.
- Handle all exceptions in termination handlers.

#### Example — Implicit Call to `terminate`

```
#include <stdexcept>
int main(){   // Noncompliant
  try {
    // program code
  } catch (std::runtime_error& e) {
    // Handle runtime errors
  } catch (std::logic_error& e) {
    // Handle logic errors
  } catch (std::exception& e) {
    // Handle all expected exceptions
  }
  return 0;
}
```

In this example, `main()` handles specific types of exceptions. An unexpected exception remains unhandled, resulting in an implicit call to the function `terminate` that terminates the program abruptly. Because `main()` calls `terminate` implicitly, Polyspace raises this defect.

#### Correction — Handle Unexpected Exceptions

One possible correction is to include a `catch(...)` block to handle unexpected exceptions so that the program can exit gracefully.

```
#include <stdexcept>
[[noreturn]] void gracefulExit(){
    // unwind stack and report errors
    std::terminate();
}
int main()   // Compliant
{
  try {
    // program code
  } catch (std::runtime_error& e) {
    // Handle runtime errors
  } catch (std::logic_error& e) {
    // Handle logic errors
  } catch (std::exception& e) {
    // Handle all expected exceptions
  }
  catch(...){
      //Exit gracefully
      gracefulExit();
  }
  return 0;
}
```

#### Example — Unhandled Exceptions in Termination Handlers

The termination handler `atexit_handler` raises an uncaught exception. The function `atexit_handler` executes after the main finishes execution. Unhandled exceptions in this function cannot be handled elsewhere, leading to an implicit call to `std::terminate()`. Polyspace flags the function.

```
#include <stdexcept>
void atexit_handler(){//Noncompliant
    throw std::runtime_error("Error in atexit function");
}
void main(){
    try{
        //...
        std::atexit(atexit_handler);
    }catch(...){

    }
}
```

**Correction — Handle All Exceptions in Termination Handlers**

To correct the issue, use a `catch(...)` block to handle all exceptions in the termination handler `atexit_handler`.

```
#include <stdexcept>
void atexit_handler(){
    try{
        //..
        throw std::runtime_error("Error in atexit function");
    }catch(...){
        //...
    }
}
void main(){
    try{
        //...
        std::atexit(atexit_handler);
    }catch(...){

    }
}
```

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR50-CPP

# CERT C++: ERR51-CPP

Handle all exceptions

## Description

### Rule Definition

*Handle all exceptions.*

### Polyspace Implementation

The rule checker checks for **Unhandled exception not caught**.

## Examples

### Unhandled exception not caught

#### Issue

The checker shows a violation if there is no `try`/`catch` in the `main` function or the `catch` block does not handle all exceptions by using a `catch(...)` block. The rule is not checked if a `main` function does not exist.

The checker does not determine if an exception of an unhandled type actually propagates to `main`.

#### Risk

Depending on the hardware and software that you use, unhandled exceptions might result in a call to `std::abort()`, which aborts program execution without deleting the variables in the stack. Such an abnormal termination results in memory leaks and security vulnerabilities.

#### Fix

Avoid unhandled exceptions. For instance, execute the operations of `main()` or task main functions in a `try-catch` block. In the catch blocks:

- Handle exceptions of type `std::exception` explicitly in appropriate catch blocks.
- Handle the base class of exceptions arising from third-party libraries.
- Handle unexpected exceptions in a `catch(...)` block.

#### Example — Exceptions Might Remain Unhandled

```
#include <stdexcept>
int main(){   // Noncompliant
  try {
    // program code
  } catch (std::runtime_error& e) {
    // Handle runtime errors
  } catch (std::logic_error& e) {
    // Handle logic errors
  } catch (std::exception& e) {
    // Handle all expected exceptions
  }
  return 0;
}
```

In this example, `main()` handles specific types of exceptions, but lacks a `catch(...)` block. An unexpected exception remains unhandled. Because `main()` does not handle *all* exceptions, Polyspace raises this defect.

#### Correction

One possible correction is to include a `catch(...)` block to handle unexpected exceptions.

```
#include <stdexcept>
int main(){   // Compliant
  try {
    // program code
  } catch (std::runtime_error& e) {
    // Handle runtime errors
  } catch (std::logic_error& e) {
    // Handle logic errors
  } catch (std::exception& e) {
    // Handle all expected exceptions
  }
  catch(...){
      //Exit gracefully
  }
  return 0;
}
```

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR51-CPP

# CERT C++: ERR52-CPP

Do not use setjmp() or longjmp()

## Description

### Rule Definition

*Do not use setjmp() or longjmp().*

### Polyspace Implementation

The rule checker checks for **Use of setjmp/longjmp**.

## Examples

### Use of setjmp/longjmp

#### Issue

**Use of setjmp/longjmp** occurs when you use a combination of `setjmp` and `longjmp` or `sigsetjmp` and `siglongjmp` to deviate from normal control flow and perform non-local jumps in your code.

#### Risk

Using `setjmp` and `longjmp`, or `sigsetjmp` and `siglongjmp` has the following risks:

- Nonlocal jumps are vulnerable to attacks that exploit common errors such as buffer overflows. Attackers can redirect the control flow and potentially execute arbitrary code.
- Resources such as dynamically allocated memory and open files might not be closed, causing resource leaks.
- If you use `setjmp` and `longjmp` in combination with a signal handler, unexpected control flow can occur. POSIX does not specify whether `setjmp` saves the signal mask.
- Using `setjmp` and `longjmp` or `sigsetjmp` and `siglongjmp` makes your program difficult to understand and maintain.

#### Fix

Perform nonlocal jumps in your code using `setjmp/longjmp` or `sigsetjmp/siglongjmp` only in contexts where such jumps can be performed securely. Alternatively, use POSIX threads if possible.

In C++, to simulate throwing and catching exceptions, use standard idioms such as `throw` expressions and `catch` statements.

#### Example - Use of `setjmp` and `longjmp`

```
#include <setjmp.h>
#include <signal.h>

extern int update(int);
extern void print_int(int);

static jmp_buf env;
```

```
void sighandler(int signum) {
    longjmp(env, signum); //Noncompliant
}
void func_main(int i) {
    signal(SIGINT, sighandler);
    if (setjmp(env)==0) { //Noncompliant
        while(1) {
            /* Main loop of program, iterates until SIGINT signal catch */
            i = update(i);
        }
    } else {
        /* Managing longjmp return */
        i = -update(i);
    }

    print_int(i);
    return;
}
```

In this example, the initial return value of `setjmp` is 0. The `update` function is called in an infinite `while` loop until the user interrupts it through a signal.

In the signal handling function, the `longjmp` statement causes a jump back to `main` and the return value of `setjmp` is now 1. Therefore, the `else` branch is executed.

### Correction — Use Alternative to `setjmp` and `longjmp`

To emulate the same behavior more securely, use a `volatile` global variable instead of a combination of `setjmp` and `longjmp`.

```
#include <setjmp.h>
#include <signal.h>

extern int update(int);
extern void print_int(int);

volatile sig_atomic_t eflag = 0;

void sighandler(int signum) {
     eflag = signum;                   /* Fix: using global variable */
}

void func_main(int i) {
      /* Fix: Better design to avoid use of setjmp/longjmp */
    signal(SIGINT, sighandler);
    while(!eflag) {                   /* Fix: using global variable */
        /* Main loop of program, iterates until eflag is changed */
        i = update(i);
    }

    print_int(i);
    return;
}
```

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR52-CPP

# CERT C++: ERR53-CPP

Do not reference base classes or class data members in a constructor or destructor function-try-block handler

## Description

### Rule Definition

*Do not reference base classes or class data members in a constructor or destructor function-try-block handler.*

### Polyspace Implementation

The rule checker checks for **Constructor or destructor function-try-block handler references base classes or class data members**.

## Examples

### Constructor or destructor function-try-block handler references base classes or class data members

#### Issue

The issue occurs when handlers of a function-try-block implementation of a class constructor or destructor references non-static members from this class or its bases.

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR53-CPP

# CERT C++: ERR54-CPP

Catch handlers should order their parameter types from most derived to least derived

## Description

### Rule Definition

*Catch handlers should order their parameter types from most derived to least derived.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Exception handlers not ordered from most-derived to base class**.
- **Incorrect order of ellipsis handler**.

## Examples

### Exception handlers not ordered from most-derived to base class

#### Issue

The issue occurs when you provide multiple handlers in a single try-catch statement or function-try-block for a derived class and some or all of its bases, and the handlers are not ordered from most-derived to base class.

### Incorrect order of ellipsis handler

#### Issue

The issue occurs when you provide multiple handlers in a single try-catch statement or function-try-block, and the ellipsis (catch-all) handler does not occur last.

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR54-CPP

# CERT C++: ERR55-CPP

Honor exception specifications

## Description

### Rule Definition

*Honor exception specifications*

### Polyspace Implementation

The rule checker checks for **Noexcept functions exiting with exception**.

## Examples

### Noexcept functions exiting with exception

#### Issue

This defect occurs when a `noexcept` entity might exit with an exception. The compiler omits the exception handing process for `noexcept` entities. When such an entity exits with an exception, the exception becomes unhandled, leading to abnormal program termination.

When a `noexcept` entity invokes other callable entities, Polyspace makes certain assumptions to calculate whether there might be unhandled exceptions.

- Function: When a `noexcept` function calls another function, Polyspace checks whether the called function might raise an exception only if it is specified as `noexcept(<false>)`. If the called function is specified as `noexcept`, Polyspace assumes that it does not raise an exception. Some standard library functions, such as the constructor of `std::string`, use pointers to functions to perform memory allocation, which might raise exceptions. Because these functions are not specified as `noexcept(<false>)`, Polyspace does not flag a function that calls these standard library functions.
- External function: When a `noexcept` function calls an external function, Polyspace flags the function declaration if the external function is specified as `noexcept(<false>)`.
- Virtual function: When a function calls a virtual function, Polyspace flags the function declaration if the virtual function is specified as `noexcept(<false>)` in a derived class. For instance, if a `noexcept` function calls a virtual function that is declared as `noexcept(<true>)` in the base class, and `noexcept(<false>)` in a subsequent derived class, Polyspace flags the declaration of the `noexcept` function.
- Pointers to function: When a `noexcept` function invokes a pointer to a function, Polyspace assumes that the pointer to the function does not raise exceptions.

When analyzing whether a function raises unhandled exceptions, Polyspace ignores:

- Exceptions raised in destructors
- Exceptions raised in `atexit()` operations

Polyspace also ignores the dynamic context when checking for exceptions. For instance, a function might raise unhandled exceptions only in certain dynamic contexts. Polyspace flags such a function even if the exception might not be raised.

#### Risk

If a `noexcept` function exits with an exception, the compiler invokes `std::terminate()` implicitly. The function `std::terminate()` terminates the program execution in an implementation-defined manner. That is, the exact process of program termination depends on the particular set of software and hardware that you use. For instance, `std:terminate()` might invoke `std::abort()` to abnormally abort the execution without unwinding the stack, leading to resource leak and security vulnerabilities.

#### Fix

Specify a function as `noexcept` or `noexcept(true)` only when you know that the function does not exit with an exception. If you are not sure, specify it by using `noexcept(false)`

#### Example

Consider this code where two functions are specified as `noexcept`. Polyspace statically analyzes these functions and the functions that they call.

```
#include <stdexcept>
#include <typeinfo>
bool f(bool flag){
    if(flag==true)
    throw flag;
    return flag;

}
void LibraryFunc_noexcept_false() noexcept(false);
void SpecFalseCT() noexcept  // Noncompliant
{
    try {
        LibraryFunc_noexcept_false();
    } catch (int &e) {
        LibraryFunc_noexcept_false();
    } catch (std::exception &e) {
    } catch (...) {
    }
}
bool flag = false;
void Caller() noexcept {          //Noncompliant
    try {
        if(f(flag)){
            //...
        }
    } catch (int i) {
        //...
    }
}
```

- Polyspace flags the `noexcept` function `SpecFaleCT()` because this function calls the `noexcept(false)` external function `LibraryFunc_noexcept_false()` without handling any exceptions that can be raised from it. These exceptions can cause the `noexcept` function to exit with an exception.

- Polyspace flags the noexcept function Caller because this function calls the noexcept(false) function f(), which contains an explicit throw statement. Even though the throw statement is not executed when flag is false, Polyspace ignores the dynamic context and flags Caller.

**Correction**

When defining functions, specify them as noexcept only when all possible exceptions are handled within the function. Otherwise, specify them as noexcept(false). In cases where an exception is not raised in the dynamic context, justify this defect by using comments.

```
#include <stdexcept>
#include <typeinfo>
bool f(bool flag){
    if(flag==true)
    throw flag;
    return flag;

}
void LibraryFunc_noexcept_false() noexcept(false);
void SpecFalseCT() noexcept(false)// Compliant
{
    try {
        LibraryFunc_noexcept_false();
    } catch (int &e) {
        LibraryFunc_noexcept_false();
    } catch (std::exception &e) {
    } catch (...) {
    }
}
bool flag = false;
void Caller() noexcept{//Noncompliant // polyspace CERT-CPP:ERR55-CPP
//[Justified:Unset] "Exception is not thrown when flag is false"
    try {
        if(f(flag)){
            //...
        }
    } catch (int i) {
        //...
    }
}
```

- The function SpecFalseCT is now specified as noexcept(false) because it calls an external function that can raise exceptions. This function is compliant with this rule.
- The function f() does not raise an exception when flag is false. The function Caller honors its exception specification, but Polyspace flags it because Polyspace ignores dynamic context. This defect is justified by using a comment.

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2020b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR55-CPP

# CERT C++: ERR56-CPP

Guarantee exception safety

## Description

### Rule Definition

*Guarantee exception safety*

### Polyspace Implementation

The rule checker checks for **Exception violating class invariant**.

## Examples

### Exception violating class invariant

#### Issue

This defect occurs when a non-`noexcept` member function attempts to raise an exception after modifying any of the fields of the class. For instance, Polyspace flags a `throw()` statement or a `new` statement if they are used after modifying the internal state of a class.

#### Risk

To facilitate recovery from an exception while preserving the invariant of the relevant objects, you must design programs to have exception safety. The C++ standard allows These levels of exception safety:

- Basic Exception Safety: This level of exception safety requires that after raising an exception, the basic invariants of all objects are maintained in a valid state and no memory is leaked. If an operation has basic exception safety, then you can destroy or assign to an object after the operations, even if an exception has been raised. The operations in standard library offers at least basic exception safety.
- Strong Exception Safety: This level of exception safety requires that after raising an exception, the state of the program remains as it was before the exception. Operations with strong exception safety either succeed or exit with an exception and have no effects on the program.
- `nothrow`: This level of safety requires that an exception cannot occur in an operation.

Without at least the basic exception safety, exceptions might corrupt the state of a program, leave the program in an invalid state, or create memory leaks. Code that does not provide at least the basic exception safety guarantee is unsafe and defective.

#### Fix

To provide at least the basic exception safety in a function, modify the class invariant only when there can be no exceptions. When performing actions that might raise exceptions, use temporary objects that do not modify the original objects invariant.

#### Example

```
#include <cstdint>
#include <cstring>
```

```
template<typename T=int>
class myArray
{
public:
    myArray(){/*...*/}
    myArray(const myArray& rhs)
    {
        DeepCopyNC(rhs);
    }
    ~myArray()
    {
        delete[] array;
    }
    void DeepCopyNC(const myArray& rhs) // Noncompliant
    {
        if (this != &rhs) {
            delete[] array;
            array = nullptr;
            len = rhs.len;

            if (len > 0) {

                array = new T[len];
                std::memcpy(array, rhs.array, len * sizeof(T));
            }
        }
    }

private:
    T* array;
    std::size_t len;
};
extern     myArray<> c1{};
void foo(){

    myArray<> c2{c1};

}
```

This example shows a generic class template `myArray` that manages a raw array. The copy constructor of this class shows two implementation of deep copy. In the function `DeepCopyNC()`, the memory operations are performed after modifying the `this->array` and `this->len` fields. If the memory operation exits with an exception, the invariant of the original class is violated. Because the class invariant is violated by an exception, the function `DeepCopyNC()` cannot guarantee basic exception safety. Polyspace flags it.

#### Correction

To guarantee basic exception safety, modify the class invariant only when the memory operation succeeds with no exceptions, as shown in `DeepCopy`.

```
#include <cstdint>
#include <cstring>
template<typename T=int>
class myArray
{
public:
    myArray(){/*...*/}
    myArray(const myArray& rhs)
    {
        DeepCopy(rhs);
    }
    ~myArray()
    {
        delete[] array;
    }
    void DeepCopy(const myArray& rhs) // Compliant
    {
        T* eTmp = nullptr;

        if (rhs.len > 0) {
            eTmp = new T[rhs.len];
            std::memcpy(eTmp, rhs.array, rhs.len * sizeof(T));
        }

        delete[] array;
        array = eTmp;
        len = rhs.len;
    }

private:
    T* array;
    std::size_t len;
};
extern     myArray<> c1{};
void foo(){

    myArray<> c2{c1};

}
```

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2022a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR56-CPP

# CERT C++: ERR57-CPP

Do not leak resources when handling exceptions

## Description

### Rule Definition

*Do not leak resources when handling exceptions.*[19]

### Polyspace Implementation

The rule checker checks for these issues:

- **Resource leak caused by exception**
- **Object left in partially initialized state**
- **Bad allocation in constructor**

## Examples

### Resource leak caused by exception

#### Issue

**Resource leak caused by exception** occurs when a function raises an unhandled exception by using a `throw` statement but does not deallocate the resources that were allocated before the exception.

#### Risk

When a function raises an unhandled exception it immediately goes out of scope. If the function manages resources and they are not deallocated prior to raising the exception, the resource is leaked. Consider this code:

```
FILE* FilePtr;
//...
void foo(){
    FilePtr = fopen("some_file.txt", "r");
```

[19] *This software has been created by MathWorks incorporating portions of: the "SEI CERT-C Website," © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, "SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition," © 2016 Carnegie Mellon University, and "SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition" © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

```
//...
    if(/*error condition*/)
    throw ERROR_CODE;
    //...
    fclose(FilePtr);
}
```

The allocated file pointer is intended to be deallocated before the function finishes execution. When an exception takes place, the function exits without deleting the pointer, which results in a resource leak.

**Fix**

To fix this defect, a function must set all resources that it allocates to a valid state before it goes out of scope. In the preceding code example, the function must delete the pointer `FilePtr` before the `throw` statement.

Instead of manually tracking the allocation and deallocation of resources, the best practice is to follow either the Resource Acquisition Is Initialization (RAII) or the Constructor Acquires, Destructor Releases (CADre) design patterns. Resource allocation is performed in constructors and resource deallocation is performed in destructors. The lifecycle of resources are controlled by scope-bound objects. When functions reach the end of their scope, the acquired resources are properly released. Consider this code:

```
void releaseFile(std::FILE* fp) { std::fclose(fp); }
std::unique_ptr<std::FILE, decltype(&releaseFile)> FilePtr;
//...
void foo(){
    FilePtr(std::fopen("some_file.txt"),&releaseFile);
//...
    if(/*error condition*/)
    throw ERROR_CODE;
}
```

The unique pointer `FilePTR` invokes the function `releaseFile` to delete the allocated resource once the function `foo` reaches the end of its scope. Whether the function exits normally with an unhandled exception, the allocated resources are deallocated.

C++ smart pointers such as `std::unique_ptr` and `std::shared_ptr` follow the RAII pattern. They simplify managing the lifecycle of resources during exception handling. Whenever possible, avoid using raw pointers.

**Example — Resource Leak Caused by Exception**

```
#include <cstdint>
#include <memory>
#include <stdexcept>
extern int sensorFlag() noexcept;
namespace Noncompliant{
    void func(){
        int* intPtr = new int;
        int data = sensorFlag();
        if(data==-1)//Error
        throw std::runtime_error("Unexpected value");//Noncompliant
        //...
        delete intPtr;
    }
}
```

In this example, the function `Noncompliant::func()` manages the raw pointer `inPtr`. The function allocates memory for it, and then releases the memory after some operations. The function exits with an exception when `data` is `-1`. In this case, the function exits before releasing the allocated memory, resulting in a memory leak. Polyspace flags the `throw` statement.

#### Correction — Deallocate Resources Before `throw` Statements

To prevent memory leak, the allocated memory must be released before raising the exception, as shown in `Compliant::func`.

The best practice is to follow the RAII design pattern. For instance, when C++14 is available, use `unique_ptr` instead of a raw pointer. `BestPractice::func` shows an implementation of `func` that follows the RAII pattern. The memory lifecycle is managed by the object itself. That is, once `func` is out of scope, the smart pointer `intPtr` deletes itself and releases the memory. Because the memory management is performed correctly by the smart pointer, `BestPractice::func` is simpler and safer.

```
#include <cstdint>
#include <memory>
#include <stdexcept>

extern int sensorFlag() noexcept;
 namespace Compliant{
    void func(){
        int* intPtr = new int;
        int data = sensorFlag();
        if(data==-1){//Error
            delete intPtr;
            throw std::runtime_error("Unexpected value");//Compliant
        }
        //...
        delete intPtr;
    }
}
namespace BestPractice{// C++14
    void func(){
        std::unique_ptr<int> intPtr = std::make_unique<int>();
        int data = sensorFlag();
        if(data==-1){//Error
            throw std::runtime_error("Unexpected value");//Compliant
        }
        //...

    }
}
```

### Object left in partially initialized state

#### Issue

**Object left in partially initialized state** occurs when a `noexcept(false)` constructor raises an unhandled exception but does not deallocate the resources that were allocated before the exception. This issue is detected only in classes that your code uses.

#### Risk

A constructor goes out of scope when it raises an unhandled exception. If the constructor manages resources and they are not deallocated prior to raising the exception, the object is left in a partially initialized state. This behavior is undefined and can produce unexpected results.

#### Fix

To fix this defect, keep track of the allocated resources and deallocate them before raising exception.

Instead of manually tracking the allocation and deallocation of resources, the best practice is to follow either the Resource Acquisition Is Initialization (RAII) or the Constructor Acquires, Destructor Releases (CADre) design patterns. Resource allocation is performed in constructors and resource deallocation is performed in destructors. The lifecycle of resources are controlled by scope-bound objects. When functions reach the end of their scope, the acquired resources are properly released. Consider this code:

```
class complex_ptr{
    complex_ptr() = default;
    ~complex_ptr() = default;
    private:
    std::unique_ptr<std::complex<double> > z;

};
```

The class `complex_ptr` uses the implicit default constructor because the resource management is performed by the smart pointer class `unique_ptr`. The default constructor does not raise exceptions and the object is not left in a partially initialized state.

C++ smart pointers such as `std::unique_ptr` and `std::shared_ptr` follow the RAII pattern. They simplify managing the lifecycle of resources during exception handling. Whenever possible, avoid using raw pointers.

#### Example — Partially Constructed Object Caused by Exceptions

```
##include<cstdlib>
#include<exception>

class complex_ptr{

    complex_ptr(){
        real = (double*)malloc(sizeof(double));
        imag = (double*)malloc(sizeof(double));
        if(real==nullptr || imag==nullptr){
            throw std::exception(); //Noncompliant
        }
    }
    ~complex_ptr(){
        free(real);
        free(imag);
    }
    private:
    double* real;
    double* imag;

};
```

```
void foo(void){
    complex_ptr Z;
    //...
}
```

In this example, the class `complex_ptr` is responsible for allocating and deallocating two raw pointers to `double`. The constructor `complex_ptr::complex_ptr()` terminates with an exception when a memory allocation operation fails. The constructor exits without deallocating the allocated resources, resulting in a partially constructed object. Polyspace flags the `throw` statement in the constructor.

#### Correction — Deallocate Resources Before raising Exceptions in Constructors

To correct this defect, deallocate the allocated resources before raising exceptions in constructor. In this code, before raising the exception, the constructor deallocates the allocated memory by calling `deallocate()`. This constructor is compliant with this rule.

```
#include<cstdlib>
#include<exception>

class complex_ptr{

    complex_ptr(){
        real = (double*)malloc(sizeof(double));
        imag = (double*)malloc(sizeof(double));
        if(real==nullptr || imag==nullptr){
            deallocate();
            throw std::exception();  //Compliant
        }
    }
    void deallocate(){
        free(real);
        free(imag);
    }
    ~complex_ptr(){
        deallocate();
    }
    private:
    double* real;
    double* imag;

};

void foo(void){
    complex_ptr Z;
    //...
}
```

### Bad allocation in constructor

#### Issue

**Bad allocation in constructor** occurs when a `new` operation is performed in a constructor without using the argument `std::nothrow` or outside exception handling blocks such as `try` or `function-try`.

**Risk**

The new operations might fail and raise a std::bad_alloc exception. If these statements are not enclosed in a try or function-try block, the exception might cause an abrupt termination of a constructor. Such an abrupt termination might leave the object in a partially constructed state, which is undefined behavior in the C++ standard.

**Fix**

When using the new operator, enclose it in a try or function-try block.

**Example — Bad Allocation in Constructors**

```
#include<cstdlib>
#include <stdexcept>
#include <new>
 class complex_ptr{

    complex_ptr(): real(new double), imag(new double){ //Noncompliant

    }
    ~complex_ptr(){
        delete real;
        delete imag;
    }
    private:
    double* real;
    double* imag;

};
void foo(void){
    complex_ptr Z;
    //...
}
```

In this example, the constructor of complex_ptr performs new operations that might raise exceptions. Because the constructor has no mechanism for handling these exceptions, they might cause the constructor to abruptly terminate. Such termination might leave the object in partially defined state because the allocated resources are not deallocated. Polyspace flags the constructor.

**Correction — Handle Exceptions Arising from new Operations in Constructors**

To correct this defect, perform the new operation in a try or function-try block.

```
#include<cstdlib>
#include <stdexcept>
#include <new>
 class complex_ptr{

    complex_ptr()try: real(new double), imag(new double){    //Compliant

    }catch(std::bad_alloc){
        //...
    }
    ~complex_ptr(){
        delete real;
        delete imag;
    }
    private:
```

```
    double* real;
    double* imag;

};
void foo(void){
    complex_ptr Z;
    //...
}
```

## Check Information

**Group:** Rule 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2021a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR57-CPP

# CERT C++: ERR58-CPP

Handle all exceptions thrown before main() begins executing

## Description

### Rule Definition

*Handle all exceptions thrown before main() begins executing.*

### Polyspace Implementation

The rule checker checks for **Exceptions raised during program startup.**

## Examples

### Exceptions raised during program startup

#### Issue

This issue occurs when an exception might arise during the construction of global and static variables before `main()` begins executing. If an exception is raised during the startup phase, you cannot write an exception handler that the compiler can execute to handle the raised exception. This exception becomes an unhandled exception. For instance, you might implement `main()` as a `function-try-catch` block to handle exceptions. None of the `catch` blocks can handle exceptions raised during the startup phase and these raised exceptions become unhandled exceptions.

When you invoke callable entities to initialize or declare global or static variables, these entities are executed during program startup. Polyspace checks whether these entities might raise exceptions during program startup by making certain assumptions.

- Function: When you call an initializer function or constructor directly to initialize a global or static variable, Polyspace checks whether the function raises an exception and flags the variable declaration if the function might raise an exception. Polyspace deduces whether a function might raise an exception regardless of its exception specification. For instance, if a `noexcept` constructor raises an exception, Polyspace flags it. If the initializer or constructor calls another function, Polyspace assumes the called function might raise an exception only if it is specified as `noexcept(<false>)`. Some standard library functions, such as the constructor of `std::string`, use pointers to functions to perform memory allocation, which might raise exceptions. Polyspace does not flag the variable declaration when these functions are used.
- External function: When you call external functions to initialize a global or static variable, Polyspace flags the declaration if the external function is specified as `noexcept(<false>)`.
- Virtual function: When you call a virtual function to initialize a global or static variable, Polyspace flags it if the virtual function is specified as `noexcept(<false>)` in any derived class. For instance, if you use a virtual initializer function that is declared as `noexcept(<true>)` in the base class, and `noexcept(<false>)` in a subsequent derived class, Polyspace flags it.
- Pointers to function: When you use a pointer to a function to initialize a global or static variable, Polyspace assumes that pointer to a function do not raise exceptions.

Polyspace ignores:

- Exceptions raised in destructors
- Exceptions raised in `atexit()` operations

Polyspace also ignores the dynamic context when checking for exceptions. For instance, you might initialize a global or static variable by using a function that raises exceptions only in a certain dynamic context. Polyspace flags such a declaration even if the exception might never be raised. You can justify such a violation by using comments in Polyspace.

**Risk**

When exceptions are not handled, the compiler might abnormally terminate the code execution without unwinding the stack depending on the set of hardware and software that you use. Consider this code where the construction of the static object `obj` might cause an exception.

```
class A{
    A(){throw(0);}
};

static A obj;

main(){
    //...
}
```

The static object `obj` is constructed by calling `A()` before `main()` starts. When `A()` raises an exception, a handler cannot be matched with the raised exception. Based on the set of software and hardware that you use, such an exception can result in program termination without stack unwinding, leading to memory leak and security vulnerabilities.

**Fix**

Avoid operations that might raise an exception in the parts of your code that might be executed before startup or after termination of the program. For instance, avoid operations that might raise exceptions in the constructor and destructor of static or global objects.

**Example**

Consider this code where the construction of the global pointer `arr` requires dynamic memory allocation.

```
#include <stdexcept>
void* alloc(size_t s) noexcept {
    return new int[s];
}
int* arr = (int*)alloc(5);//Noncompliant
int main(){
    //..
    return 0;
}
```

Dynamic memory allocation by using the `new` operator can raise an exception. Because constructing `arr` can raise an exception before `main()` begins execution, Polyspace flags the declaration.

**Correction**

Avoid operations that might raise exception when constructing global objects. For instance, you can initialize the global pointer `arr` by using a `nullptr`. Then allocate memory for `arr` in `main()` in a `try-catch` code block.

```
#include <stdexcept>
#include<vector>
void* alloc(size_t s) noexcept {
    return new int[s];
}
int* arr =nullptr;
int main(){
    try{
        arr =  (int*)alloc(5);
    }
    catch(std::bad_alloc e){
        //..
    }
    //..
    return 0;
}
```

In this case, the dynamic memory allocation operation raises the `std::bad_alloc` exception in `main()` where it can be handled by the catch blocks of code.

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2020b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR58-CPP

# CERT C++: ERR59-CPP

Do not throw an exception across execution boundaries

## Description

### Rule Definition

*Do not throw an exception across execution boundaries.*

### Polyspace Implementation

The rule checker checks for **Exceptions raised from library interface**.

## Examples

### Exceptions raised from library interface

#### Issue

This issue occurs when both of these conditions are met:

- A function is specified to be across an execution boundary.
- The function raises an exception.

Polyspace assumes that library interface functions are across an execution boundary. A violation is not raised if a `new` operator in a library interface function raises a `std::bad_alloc` exception.

To use this checker correctly, specify the library interface functions in your. Declare a function *`foo()`* as a library interface function by setting their `visibility` attribute. For instance:

- GNU and Clang compiler: `void __attribute__((visibility("default"))) foo(){/*...*/}`
- Visual Studio: `void __declspec(dllexport) foo(){/**.../}`

If you do not explicitly specify a function as visible, Polyspace assumes that it is not part of a library interface.

#### Risk

Exception handling requires interoperability between the functions that raise the exception and the functions that handle the raised exceptions. Library functions might implement exception handling by using incompatible interfaces. For instance:

```
// lib.h
void foo() noexcept(false);
//lib.cpp
void foo() noexcept(false){
  //...
  throw 42;
}
//App.cpp
#include"lib.h"
```

```
int main(){
  try{
    foo();
  }catch(int& e){
    //handle exception
  }
}
```

Say you use the library interface `lib.cpp` that has been compiled by using a GCC 4.9 compiler. Then you compile the application `App.cpp` by using Microsoft Visual Studio. These two portions of your code use incompatible exception-handling interfaces. If the library interface function raises an exception, it is not handled and terminates the application unexpectedly.

**Fix**

Avoid raising exceptions in library interfaces. Instead of raising exceptions, return different error codes to handle unexpected situations.

**Example — Exception Arising from Visible Function**

```
#include<exception>

void __attribute__((visibility("default"))) foo(int rd) noexcept(false) { //Noncompliant
    if (rd==-1)
        throw std::exception();
}
```

In this example, the `visibility` attribute of the function `foo()` is set to `default`. Because `foo()` might be used in modules that have different exception handling mechanisms, exceptions raised by `foo()` might remain unhandled. Avoid raising exceptions from visible functions functions.

**Correction — Use Other Methods to Handle Errors**

Use other methods for handling errors. For instance, set an error flag and return an error code when a logic error occurs in the code.

```
#include<exception>
int FLAG;

int __attribute__((visibility("default"))) foo(bool rd) noexcept(true) {
    if (rd==-1){
        FLAG = 52;
        return -1;//Compliant
    }
}
```

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2022b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ERR59-CPP

# CERT C++: ERR61-CPP

Catch exceptions by lvalue reference

## Description

### Rule Definition

*Catch exceptions by lvalue reference.*

### Polyspace Implementation

The rule checker checks for **Exception object initialized by copy in catch statement**.

## Examples

### Exception object initialized by copy in `catch` statement

#### Issue

The issue occurs when a `catch` statement

```
catch (exceptionType customExc) {
  //...
}
```

initializes the exception object `customExc` by copy.

#### Risk

If *`exceptionType`* has a nontrivial copy constructor or if the exception thrown belongs to a class derived from *`exceptionType`*, the copying can produce object slicing or undefined behavior.

#### Fix

Catch the exception by reference or `const` reference.

```
catch (exceptionType &customExc) {
  //...
}
```

#### Example - Derived Class Exception Caught by Value

```
#include <exception>
#include <string>
#include <typeinfo>
#include <iostream>

// Class declarations
class BaseExc {
public:
    explicit BaseExc();
    virtual ~BaseExc() {};
protected:
    BaseExc(const std::string& type);
private:
    std::string _id;
};

class IOExc: public BaseExc {
public:
    explicit IOExc();
};

//Class method declarations
BaseExc::BaseExc():_id(typeid(this).name()) {
}
BaseExc::BaseExc(const std::string& type): _id(type) {
}
IOExc::IOExc(): BaseExc(typeid(this).name()) {
}

int input(void);

int main(void) {
    int rnd = input();
    try {
        if (rnd==0) {
            throw IOExc();
        } else {
            throw BaseExc();
        }
    }


    catch(BaseExc exc) {  //Noncompliant
        std::cout << "Intercept BaseExc" << std::endl;
    }
    return 0;
}
```

In this example, the `catch` statement takes a `BaseExc` object by value. Catching exceptions by value causes copying of the object. The copying can cause:

- Undefined behavior of the exception if it fails.
- Object slicing if an exception of the derived class `IOExc` is caught.

**Correction — Catch Exceptions by Reference**

One possible correction is to catch exceptions by reference.

```
#include <exception>
#include <string>
#include <typeinfo>
#include <iostream>

// Class declarations
class BaseExc {
public:
    explicit BaseExc();
    virtual ~BaseExc() {};
protected:
    BaseExc(const std::string& type);
private:
    std::string _id;
};

class IOExc: public BaseExc {
public:
    explicit IOExc();
};

//Class method declarations
BaseExc::BaseExc():_id(typeid(this).name()) {
}
BaseExc::BaseExc(const std::string& type): _id(type) {
}
IOExc::IOExc(): BaseExc(typeid(this).name()) {
}

int input(void);

int main(void) {
    int rnd = input();
    try {
        if (rnd==0) {
            throw IOExc();
        } else {
            throw BaseExc();
        }
    }



    catch(BaseExc& exc) {
        std::cout << "Intercept BaseExc" << std::endl;
    }
    return 0;
}
```

## Check Information

**Group:** 08. Exceptions and Error Handling (ERR)

# Version History

**Introduced in R2019b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

ERR61-CPP

# CERT C++: OOP50-CPP

Do not invoke virtual functions from constructors or destructors

## Description

### Rule Definition

*Do not invoke virtual functions from constructors or destructors.*[20]

### Polyspace Implementation

The rule checker checks for **Virtual function call from constructors and destructors**.

## Examples

### Virtual function call from constructors and destructors

#### Issue

**Virtual function call from constructors and destructors** occurs when you invoke virtual functions in a constructor or a destructor with possibly unexpected results.

When you call virtual functions in the constructor or destructor of a class in a hierarchy, it is not clear which instance of the virtual function you intend to invoke. Calls to virtual functions in a constructor or a destructor resolves to the implementation of the virtual function in the currently executing class instead of the most derived override.

There are two cases where calling virtual functions from a constructor or destructor does not raise this defect.

- When you use the explicitly qualified ID to call the virtual function. Consider this code:

  ```
  Base(){
      Base::foo();
  }
  ```

  The call to `Base::foo` uses the explicitly qualified ID of the function. This call is compliant with this rule because it explicitly states that the implementation of `foo` belonging to `Base` is invoked.

[20] *This software has been created by MathWorks incorporating portions of: the “SEI CERT-C Website,” © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, ”SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition,” © 2016 Carnegie Mellon University, and “SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition” © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

- When the you specify the called virtual function as `final` in the currently executing class. Consider this code:

  ```
  Base(){
      foo();
  }
  //...
  void foo() override final{
  //...
  }
  ```

  In this case, a call to `foo` implies a call to `Base::foo` because the function is specified as the final override.

**Risk**

When you call a virtual function, you expect the compiler to resolve the call to the most derived override of the virtual function at runtime. Unlike in other functions, virtual function calls in constructors and destructors are resolved differently, resulting in unexpected behavior. Invoking virtual functions in constructor and destructors might cause undefined behavior, resulting in a memory leak and security vulnerabilities. Consider this code:

```
#include <iostream>

class Base
{
public:
    Base() { foo(); }  //Noncompliant
    ~Base(){bar();}    //Noncompliant
    virtual void foo() {
        std::cout<<"Base Constructor\n";
    }
    virtual void bar(){
        std::cout<<"Base Destructor\n";
    }

};
class Derived : public Base
{
public:
    Derived() : Base() {}
    ~Derived() = default;
    virtual void foo() {
        std::cout<<"Derived constructor\n";
    }
    virtual void bar() {
        std::cout<<"Derived Constructor\n";
    }
};
int main(){
    Derived d;
    return 1;
}
```

The constructor of `d` calls the constructor for `Base` class, which invokes the virtual function `foo`. Because the derived class is not constructed yet, the compiler cannot invoke `Derived::foo`. Only the function `Base::foo()` is invoked. Similarly, when the virtual function `bar` is invoked in the destructor of `Base`, the derived class `Derived` is already destroyed. The compiler cannot invoke `Derived::bar`. Only the function `Base::bar` is invoked. The output of this code is:

```
Base Constructor
Base Destructor
```

instead of:

```
Base Constructor
Derived constructor
Derived Constructor
Base Destructor
```

The portion of `d` belonging to the class `Derived` is neither allocated nor deallocated. This behavior might result in memory leaks or security vulnerabilities.

**Fix**

To fix this issue, avoid calling virtual functions in constructors and destructors. For typical constructor or destructor tasks such as memory allocation and deallocation, initialization, or message logging, use functions that are specific to each class in a hierarchy.

**Example — Class-Specific Memory Management**

```
#include <iostream>

class Base {
public:
    Base()
    {
        allocator();  //Noncompliant
    }
    virtual ~Base()
    {
        deallocator();  //Noncompliant
    }

    virtual void allocator(){
        //...
    }
    virtual void deallocator(){
        //...
    }
};

class Derived : public Base {
public:
    Derived() : Base() {}
    virtual ~Derived() = default;
protected:
    void allocator() override
    {
        Base::allocator();
        // Get derived resources...
    }
    void deallocator() override
    {
        // Release derived resources...
        Base::deallocator();
    }
};

int main(){
    Derived dObj;
    //...
    return 1;
}
```

In this example, the code attempts class-specific memory management by implementing the functions `allocator` and `deallocator` as virtual. A call to these functions does not resolve to the most derived override.

- During the construction of the `Derived` object `dObj`, only the function `Base::allocator()` is invoked. Because the `Derived` class is not constructed yet, the function `Derived::allocator` is not invoked.
- During the destruction of `dObj`, only the function `Base::deallocator` is invoked because the class `Derived` is already destroyed.

Because of the use of virtual functions in the constructor and destructor of `dObj`, the `Derived` portion of `dObj` is neither allocated nor deallocated. This behavior is unexpected and might lead to memory leaks and security vulnerabilities.

#### Correction — Class-Specific Memory Management

One possible correction is to use class-specific nonvirtual functions for tasks that are commonly performed in constructors and destructors. In this code, allocation and deallocation tasks are performed by class-specific nonvirtual functions.

```
#include <iostream>

class Base {
public:
    Base()
    {
        allocator_base();
    }
    virtual ~Base()
    {
        deallocator_base();
    }
protected:
    void allocator_base(){
        // Allocate base resources
    }
    void deallocator_base(){
        // Deallocate base resources
    }
};

class Derived : public Base {
public:
    Derived(){
        allocator_derived();
    }
    virtual ~Derived(){
        deallocator_derived();
    }
protected:
    void allocator_derived()
    {
        // Allocate derived resources...
    }
    void deallocator_derived()
    {
        // Deallocate derived resources...
    }
};

int main(){
    Derived dObj;
    //...
    return 1;
}
```

## Check Information

**Group:** Rule 09. Object Oriented Programming (OOP)

# Version History

**Introduced in R2021a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

OOP50-CPP

# CERT C++: OOP51-CPP

Do not slice derived objects

## Description

### Rule Definition

*Do not slice derived objects.*

### Polyspace Implementation

The rule checker checks for **Object slicing**.

## Examples

### Object slicing

#### Issue

**Object slicing** occurs when you pass a derived class object by value to a function, but the function expects a base class object as parameter.

#### Risk

If you pass a derived class object *by value* to a function, you expect the derived class copy constructor to be called. If the function expects a base class object as parameter:

- The base class copy constructor is called.
- In the function body, the parameter is considered as a base class object.

In C++, `virtual` methods of a class are resolved at run time according to the actual type of the object. Because of object slicing, an incorrect implementation of a `virtual` method can be called. For instance, the base class contains a `virtual` method and the derived class contains an implementation of that method. When you call the `virtual` method from the function body, the base class method is called, even though you pass a derived class object to the function.

#### Fix

One possible fix is to pass the object by reference or pointer. Passing by reference or pointer does not cause invocation of copy constructors. If you do not want the object to be modified, use a `const` qualifier with your function parameter.

Another possible fix is to overload the function with another function that accepts the derived class object as parameter.

#### Example - Function Call Causing Object Slicing

```
#include <iostream>

class Base {
public:
    explicit Base(int b) {
        _b = b;
    }
    virtual ~Base() {}
    virtual int update() const;
protected:
    int _b;
};


class Derived: public Base {
public:
    explicit Derived(int b):Base(b) {}
    int update() const;
};

//Class methods definition

int Base::update() const {
    return (_b + 1);
}

int Derived::update() const {
    return (_b -1);
}


//Other function definitions
void funcPassByValue(const Base bObj) {
    std::cout << "Updated _b=" << bObj.update() << std::endl;
}

int main() {
    Derived dObj(0);
    funcPassByValue(dObj);       //Function call slices object //Noncompliant
    return 0;
 }
```

In this example, the call `funcPassByValue(dObj)` results in the output `Updated _b=1` instead of the expected `Updated _b=-1`. Because `funcPassByValue` expects a `Base` object parameter, it calls the `Base` class copy constructor.

Therefore, even though you pass the `Derived` object `dObj`, the function `funcPassByValue` treats its parameter `b` as a `Base` object. It calls `Base::update()` instead of `Derived::update()`.

**Correction — Pass Object by Reference or Pointer**

One possible correction is to pass the `Derived` object `dObj` by reference or by pointer. In the following, corrected example, `funcPassByReference` and `funcPassByPointer` have the same objective as `funcPassByValue` in the preceding example. However, `funcPassByReference` expects a reference to a `Base` object and `funcPassByPointer` expects a pointer to a `Base` object.

Passing the `Derived` object `d` by a pointer or by reference does not slice the object. The calls `funcPassByReference(dObj)` and `funcPassByPointer(&dObj)` produce the expected result `Updated _b=-1`.

```
#include <iostream>

class Base {
```

```
public:
    explicit Base(int b) {
        _b = b;
    }
    virtual ~Base() {}
    virtual int update() const;
protected:
    int _b;
};


class Derived: public Base {
public:
    explicit Derived(int b):Base(b) {}
    int update() const;
};

//Class methods definition

int Base::update() const {
    return (_b + 1);
}

int Derived::update() const {
    return (_b -1);
}


//Other function definitions
void funcPassByReference(const Base& bRef) {
    std::cout << "Updated _b=" << bRef.update() << std::endl;
}

void funcPassByPointer(const Base* bPtr) {
    std::cout << "Updated _b=" << bPtr->update() << std::endl;
}

int main() {
    Derived dObj(0);
    funcPassByReference(dObj);       //Function call does not slice object
    funcPassByPointer(&dObj);       //Function call does not slice object
    return 0;
 }
```

**Note** If you pass by value, because a copy of the object is made, the original object is not modified. Passing by reference or by pointer makes the object vulnerable to modification. If you are concerned about your original object being modified, add a `const` qualifier to your function parameter, as in the preceding example.

## Check Information

**Group:** 09. Object Oriented Programming (OOP)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

OOP51-CPP

# CERT C++: OOP52-CPP

Do not delete a polymorphic object without a virtual destructor

## Description

### Rule Definition

*Do not delete a polymorphic object without a virtual destructor.*

### Polyspace Implementation

The rule checker checks for **Base class destructor not virtual**.

## Examples

### Base class destructor not virtual

#### Issue

**Base class destructor not virtual** occurs when a class has `virtual` functions but not a `virtual` destructor.

#### Risk

The presence of `virtual` functions indicates that the class is intended for use as a base class. However, if the class does not have a `virtual` destructor, it cannot behave polymorphically for deletion of derived class objects.

If a pointer to this class refers to a derived class object, and you use the pointer to delete the object, only the base class destructor is called. Additional resources allocated in the derived class are not released and can cause a resource leak.

#### Fix

One possible fix is to always use a `virtual` destructor in a class that contains `virtual` functions.

#### Example - Base Class Destructor Not Virtual

```
class Base {         //Noncompliant
        public:
                Base(): _b(0) {};
                virtual void update() {_b += 1;};
        private:
                int _b;
};

class Derived: public Base {
        public:
                Derived(): _d(0) {};
                ~Derived() {_d = 0;};
                virtual void update() {_d += 1;};
        private:
                int _d;
};
```

In this example, the class `Base` does not have a `virtual` destructor. Therefore, if a `Base*` pointer points to a `Derived` object that is allocated memory dynamically, and the `delete` operation is performed on that `Base*` pointer, the `Base` destructor is called. The memory allocated for the additional member `_d` is not released.

The defect appears on the base class definition. Following are some tips for navigating in the source code:

- To find classes derived from the base class, right-click the base class name and select **Search For All References**. Browse through each search result to find derived class definitions.
- To find if you are using a pointer or reference to a base class to point to a derived class object, right-click the base class name and select **Search For All References**. Browse through search results that start with `Base*` or `Base&` to locate pointers or references to the base class. You can then see if you are using a pointer or reference to point to a derived class object.

#### Correction — Make Base Class Destructor Virtual

One possible correction is to declare a `virtual` destructor for the class `Base`.

```
class Base {
        public:
                Base(): _b(0) {};
                virtual ~Base() {_b = 0;};
                virtual void update() {_b += 1;};
        private:
                int _b;
};

class Derived: public Base {
        public:
                Derived(): _d(0) {};
                ~Derived() {_d = 0;};
                virtual void update() {_d += 1;};
        private:
                int _d;
};
```

## Check Information

**Group:** 09. Object Oriented Programming (OOP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

OOP52-CPP

# CERT C++: OOP53-CPP

Write constructor member initializers in the canonical order

## Description

### Rule Definition

*Write constructor member initializers in the canonical order.*

### Polyspace Implementation

The rule checker checks for **Members not initialized in canonical order**.

## Examples

### Members not initialized in canonical order

#### Description

**Members not initialized in canonical order** occurs when the initializer list of a class constructor:

- Does not initialize data members of the class in the order in which they are declared.

  For instance:

  ```
  class aClass {
     int var1;
     int var2;
  public:
     aClass(int val): var2(val), var1(val) {}
  };
  ```
- Does not call base class constructors in the order in which they appear in the base-specifier list.

  For instance:

  ```
  class aClass: baseClass1, baseClass2 {
     aClass(int val): baseClass2(val), baseClass1(val) {}
  }
  ```

#### Risk

The order in which data members or base class constructors appear in the initializer list does not reflect the actual order of initialization. Data members are initialized in the order of declaration and base class constructors are called in the order in which they appear in the base-specifier list.

However, you or another developer can mistake the order in the initializer list as the actual initialization order. As a result, you might introduce dependencies between the initializations that results in reading an uninitialized region of memory. For instance, this initializer list might indicate that `bVar` is first initialized with the constructor argument `x` and then `aVar` is initialized with `bVar`:

```
class aClass {
```

```
   int aVar;
   int bVar;
public:
   aClass(int x): bVar(x), aVar(bVar) {}
};
```

However, the initialization happens in the order of declaration and an uninitialized `bVar` is read first.

#### Fix

In the initializer list of a class constructor:

- Specify class data members in the same order as you declare them in the class

  For instance:

  ```
  class aClass {
     int var1;
     int var2;
  public:
     aClass(int val): var1(val), var2(val) {}
  };
  ```
- Call base constructors in the same order as you specify them in the base-specifier list.

  For instance:

  ```
  class aClass: baseClass1, baseClass2 {
     aClass(int val): baseClass1(val), baseClass2(val) {}
  }
  ```

## Check Information

**Group:** 09. Object Oriented Programming (OOP)

# Version History

**Introduced in R2020a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

OOP53-CPP

# CERT C++: OOP54-CPP

Gracefully handle self-copy assignment

## Description

### Rule Definition

*Gracefully handle self-copy assignment.*

### Polyspace Implementation

The rule checker checks for **Self assignment not tested in operator**.

## Examples

### Self assignment not tested in operator

#### Issue

**Self assignment not tested in operator** occurs when you do not test if the argument to the copy assignment operator of an object is the object itself.

#### Risk

Self-assignment causes unnecessary copying. Though it is unlikely that you assign an object to itself, because of aliasing, you or users of your class cannot always detect a self-assignment.

Self-assignment can cause subtle errors if a data member is a pointer and you allocate memory dynamically to the pointer. In your copy assignment operator, you typically perform these steps:

1. Deallocate the memory originally associated with the pointer.

   ```
   delete ptr;
   ```
2. Allocate new memory to the pointer. Initialize the new memory location with contents obtained from the operator argument.

   ```
    ptr = new ptrType(*(opArgument.ptr));
   ```

If the argument to the operator, `opArgument`, is the object itself, after your first step, the pointer data member in the operator argument, `opArgument.ptr`, is not associated with a memory location. `*opArgument.ptr` contains unpredictable values. Therefore, in the second step, you initialize the new memory location with unpredictable values.

#### Fix

Test for self-assignment in the copy assignment operator of your class. Only after the test, perform the assignments in the copy assignment operator.

#### Example - Missing Test for Self-Assignment

```
class MyClass1 { };
class MyClass2 {
public:
```

```
    MyClass2()                : p_(new MyClass1())      { }
    MyClass2(const MyClass2& f)   : p_(new MyClass1(*f.p_)) { }
    ~MyClass2()                {
        delete p_;
    }
    MyClass2& operator= (const MyClass2& f) //Noncompliant
    {
        delete p_;
        p_ = new MyClass1(*f.p_);
        return *this;
    }
private:
    MyClass1* p_;
};
```

In this example, the copy assignment operator in `MyClass2` does not test for self-assignment. If the parameter `f` is the current object, after the statement `delete p_`, the memory allocated to pointer `f.p_` is also deallocated. Therefore, the statement `p_ = new MyClass1(*f.p_)` initializes the memory location that `p_` points to with unpredictable values.

#### Correction — Test for Self-Assignment

One possible correction is to test for self-assignment in the copy assignment operator.

```
class MyClass1 { };
class MyClass2 {
public:
    MyClass2()                : p_(new MyClass1())      { }
    MyClass2(const MyClass2& f)   : p_(new MyClass1(*f.p_)) { }
    ~MyClass2()                {
        delete p_;
    }
    MyClass2& operator= (const MyClass2& f)
    {
        if(&f != this) {
           delete p_;
           p_ = new MyClass1(*f.p_);
        }
        return *this;
    }
private:
    MyClass1* p_;
};
```

## Check Information

**Group:** 09. Object Oriented Programming (OOP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

OOP54-CPP

# CERT C++: OOP55-CPP

Do not use pointer-to-member operators to access nonexistent members

## Description

### Rule Definition

*Do not use pointer-to-member operators to access nonexistent members.*[21]

### Polyspace Implementation

The rule checker checks for **Pointer to member accessing non-existent class members**

## Examples

### Pointer to Member Accessing Non-Existent Class Members

#### Issue

This issue occurs when you use of pointer-to-member operators in these instances:

- You cast a pointer-to-member type to another pointer-to-member type which does not contain the class member pointed to. For instance, in this code snippet, pointer-to-member `ptrToMember` is obtained from `myObj::f` but is then upcast to `baseObj::*` which does not contain a method `f`.

```
class baseObj {
  public:
    virtual ~baseObj() = default;
};

class myObj : public baseObj {
  public:
    void f();
};


void func() {
    baseObj* foo = new baseObj();
```

[21] *This software has been created by MathWorks incorporating portions of: the “SEI CERT-C Website,” © 2017 Carnegie Mellon University, the SEI CERT-C++ Web site © 2017 Carnegie Mellon University, ”SEI CERT C Coding Standard – Rules for Developing safe, Reliable and Secure systems – 2016 Edition,” © 2016 Carnegie Mellon University, and “SEI CERT C++ Coding Standard – Rules for Developing safe, Reliable and Secure systems in C++ – 2016 Edition” © 2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.*

*ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

*This software and associated documentation has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.*

```
    void (baseObj::*ptrToMember)() =
        static_cast<void (baseObj::*)()>(&myObj::f); //Noncompliant
    (foo->*ptrToMember)();
}
```

Polyspace does not flag the casting operation if the type being cast to inherits from the other type. For example, in the preceding code, if `baseObj` inherits from `myObj`.

- The second operand of the pointer-to-member operator is a null pointer. For instance, in this code snippet, `ptrToMember` is declared but not initialized and defaults to a null pointer.

```
class baseObj {
  public:
    virtual ~baseObj() = default;
};
static void (baseObj::*ptrToMember)(); //Not initialized

void func() {
    baseObj* foo = new baseObj();

    (foo->*ptrToMember)(); //Noncompliant
}
```

#### Risk

The use of pointer-to-member operators results in undefined behavior in these cases:

- The dynamic type of the first operand does not contain the member referred to by the second operand.
- The second operand pointer is null.

#### Fix

Do not cast a pointer-to-member to a type that does not contain the class member pointed to, and do not use a null pointer as the second operand of the pointer to member operator.

#### Example — Pointer-to-Member Accesses Non-Existent Class Member

```
class Base {
  public:
    void f() {
    }
};

class Derived : public Base {
  public:
    int myVar;
};

void func() {
    Base myObj;
    auto ptrToMethod =
        static_cast<void (Base::*)()>(&Derived::f); // Compliant
    auto ptrToMember =
        static_cast<int(Base::*)>(&Derived::myVar); // Noncompliant

    (myObj.*ptrToMethod)(); // OK, equivalent to myObj.f()
    myObj.*ptrToMember;     // Undefined behavior
}
```

In this example, Polyspace flags the casting of `ptrToMember`, which is obtained from `Derived::myVar`, to `Base::*` because this class does not contain a member `myVar`. If you execute the code, the call `myObj.*ptrToMember;` results in undefined behavior.

Polyspace does not flag the casting of `ptrToMethod` to `Base::*` because class `Derived` inherits method `f` from `Base`.

**Correction — Use Correct Type for Underlying Object**

```
class Base {
  public:
    void f() {
    }
};

class Derived : public Base {
  public:
    int myVar;
};

void func() {
    Derived myObj;
    auto ptrToMethod =
        static_cast<void (Base::*)()>(&Derived::f); // Compliant
    int (Derived::*ptrToMember) = &Derived::myVar;

    (myObj.*ptrToMethod)(); // OK, equivalent to myObj.f()
    myObj.*ptrToMember; // Compliant
}
```

One possible correction is to use the correct type for the object that you use as the first operator of the pointer-to-member operator. In the preceding code, `myObj` is of type `Derived` and the casting is no longer necessary in the declaration of `ptrToMember`.

## Check Information

**Group:** Rule 09. Object Oriented Programming (OOP)

# Version History

**Introduced in R2022a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

OOP55-CPP

# CERT C++: OOP57-CPP

Prefer special member functions and overloaded operators to C Standard Library functions

## Description

### Rule Definition

*Prefer special member functions and overloaded operators to C Standard Library functions.*

### Polyspace Implementation

The rule checker checks for **Bytewise operations on nontrivial class object**.

## Examples

### Bytewise operations on nontrivial class object

#### Issue

**Bytewise operations on nontrivial class object** occurs when you use C Standard library functions to perform bytewise operation on non-trivial or non-standard layout class type objects. For definitions of trivial and standard layout classes, see the C++ Standard, [class], paragraphs 6 and 7 respectively.

The checker raises a defect you initialize or copy non-trivial class type objects using these functions:

- `std::memset`
- `std::memcpy`
- `std::strcpy`
- `std::memmove`

Or when you compare non-standard layout class type objects using these functions:

- `std::memcmp`
- `std::strcmp`

**Bytewise operations on nontrivial class object** raises no defect if the bytewise operation is performed through an alias. For example no defect is raised in the bytewise comparison and copy operations in this code. The bytewise operations use `dptr` and `sptr`, the aliases of non-trivial or non-standard layout class objects `d` and `s`.

```
void func(NonTrivialNonStdLayout *d, const NonTrivialNonStdLayout *s)
{
   void* dptr = (void*)d;
   const void* sptr = (void*)s;
   // ...
   // ...
   // ...
   if (!std::memcmp(dptr, sptr, sizeof(NonTrivialNonStdLayout))) {
     (void)std::memcpy(dptr, sptr, sizeof(NonTrivialNonStdLayout));
      // ...
   }
}
```

### Risk

Performing bytewise comparison operations by using C Standard library functions on non-trivial or non-standard layout class type object might result in unexpected values due to implementation details. The object representation depends on the implementation details, such as the order of private and public members, or the use of virtual function pointer tables to represent the object.

Performing bytewise setting operations by using C Standard library functions on non-trivial or non-standard layout class type object can change the implementation details. The operation might result in abnormal program behavior or a code execution vulnerability. For instance, if the address of a member function is overwritten, the call to this function invokes an unexpected function.

### Fix

To perform bytewise operations non-trivial or non-standard layout class type object, use these C++ special member functions instead of C Standard library functions.

| C Standard Library Functions | C++ Member Functions |
|---|---|
| `std::memset` | Class constructor |
| `std::memcpy`<br>`std::strcpy`<br>`std::memmove` | Class copy constructor<br>Class move constructor<br>Copy assignment operator<br>Move assignment operator |
| `std::memcmp`<br>`std::strcmp` | `operator<()`<br>`operator>()`<br>`operator==()`<br>`operator!=()` |

### Example - Using `memset` with non-trivial class object

```
#include <cstring>
#include <iostream>
#include <utility>

class nonTrivialClass
{
    int scalingFactor;
    int otherData;
public:
    nonTrivialClass() : scalingFactor(1) {}
    void set_other_data(int i);
    int f(int i)
    {
        return i / scalingFactor;
    }
    // ...
};

void func()
{
    nonTrivialClass c;
    // ... Code that mutates c ...
    std::memset(&c, 0, sizeof(nonTrivialClass)); //Noncompliant
    std::cout << c.f(100) << std::endl;
}
```

In this example, `func()` uses `std::memset` to reinitialize non-trivial class object `c` after it is first initialized with its default constructor. This bytewise operation might not properly initialize the value representation of `c`.

#### Correction — Define Function Template That Uses `std::swap`

One possible correction is to define a function template `clear()` that uses `std::swap` to perform a swap operation. The call to `clear()`properly reinitializes object `c` by swapping the contents of `c` and default initialized object `empty`.

```
 #include <cstring>
#include <iostream>
#include <utility>

class nonTrivialClass
{
    int scalingFactor;
    int otherData;
public:
    nonTrivialClass() : scalingFactor(1) {}
    void set_other_data(int i);
    int f(int i)
    {
        return i / scalingFactor;
    }
    // ...
};

template <typename T>
T& clear(T& o)
{
    using std::swap;
    T empty;
    swap(o, empty);
    return o;
}

void func()
{
    nonTrivialClass c;
    // ... Code that mutates c ...

    clear(c);
    std::cout << c.f(100) << std::endl;
}
```

## Check Information

**Group:** Rule 09. Object Oriented Programming (OOP)

## Version History

**Introduced in R2019b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

OOP57-CPP

# CERT C++: OOP58-CPP

Copy operations must not mutate the source object

## Description

### Rule Definition

*Copy operations must not mutate the source object.*

### Polyspace Implementation

The rule checker checks for **Copy operation modifying source operand**.

## Examples

### Copy operation modifying source operand

#### Issue

**Copy operation modifying source operand** occurs when a copy constructor or copy assignment operator modifies a mutable data member of its source operand.

For instance, this copy constructor A modifies the data member m of its source operand other:

```
class A {
  mutable int m;

public:
 ...
  A(const A &other) : m(other.m) {
    other.m = 0; //Modification of source
  }
}
```

#### Risk

A copy operation with a copy constructor (or copy assignment operator):

```
className new_object = old_object; //Calls copy constructor of className
```

copies its source operand old_object to its destination operand new_object. After the operation, you expect the destination operand to be a copy of the unmodified source operand. If the source operand is modified during copy, this assumption is violated.

#### Fix

Do not modify the source operand in the copy operation.

If you are modifying the source operand in a copy constructor to implement a move operation, use a move constructor instead. Move constructors are defined in the C++11 standard and later.

#### Example - Copy Constructor Modifying Source

```
#include <algorithm>
```

```
#include <vector>

class A {
  mutable int m;

public:
  A() : m(0) {}
  explicit A(int m) : m(m) {}

  A(const A &other) : m(other.m) {
    other.m = 0; //Noncompliant
  }

  A& operator=(const A &other) {
    if (&other != this) {
      m = other.m;
      other.m = 0; //Noncompliant //Noncompliant
    }
    return *this;
  }

  int get_m() const { return m; }
};

void f() {
  std::vector<A> v{10};
  A obj(12);
  std::fill(v.begin(), v.end(), obj);
}
```

In this example, a vector of ten objects of type `A` is created. The `std::fill` function copies an object of type `A`, which has a data member with value 12, to each of the ten objects. After this operation, you might expect that all ten objects in the vector have a data member with value 12.

However, the first copy modifies the data member of the source to the value 0. The remaining nine copies copy this value. After the `std::fill` call, the first object in the vector has a data member with value 12 and the remaining objects have data members with value 0.

**Correction — Use Move Constructor for Modifying Source**

Do not modify data members of the source operand in a copy constructor or copy assignment operator. If you want your class to have a move operation, use a move constructor instead of a copy constructor.

In this corrected example, the copy constructor and copy assignment operator of class `A` do not modify the data member `m`. A separate move constructor modifies the source operand.

```
#include <algorithm>
#include <vector>

class A {
  int m;

public:
  A() : m(0) {}
  explicit A(int m) : m(m) {}
  A(const A &other) : m(other.m) {}
  A(A &&other) : m(other.m) { other.m = 0; }

  A& operator=(const A &other) {
    if (&other != this) {
      m = other.m;
    }
    return *this;
  }

  //Move constructor
  A& operator=(A &&other) {
    m = other.m;
    other.m = 0;
    return *this;
  }

  int get_m() const { return m; }
};

void f() {
  std::vector<A> v{10};
  A obj(12);
  std::fill(v.begin(), v.end(), obj);
}
```

## Check Information

**Group:** 09. Object Oriented Programming (OOP)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

OOP58-CPP

# CERT C++: CON33-C

Avoid race conditions when using library functions

## Description

### Rule Definition

*Avoid race conditions when using library functions.*

### Polyspace Implementation

The rule checker checks for **Data race through standard library function call**.

### Extend Checker

For the rule checker to detect issues, your code must use one of the concurrency primitives recognized by Polyspace for thread creation and shared variable protection. Otherwise, you must explicitly specify tasks, interrupts, critical sections, and other multitasking options in your project configuration. See “Multitasking”.

## Examples

### Data race through standard library function call

#### Issue

**Data race through standard library function call** occurs when:

- Multiple tasks call the same standard library function.

  For instance, multiple tasks call the `strerror` function.
- The calls are not protected using a common protection.

  For instance, the calls are not protected by the same critical section.

Functions flagged by this defect are not guaranteed to be reentrant. A function is reentrant if it can be interrupted and safely called again before its previous invocation completes execution. If a function is not reentrant, multiple tasks calling the function without protection can cause concurrency issues. For the list of functions that are flagged, see CON33-C: Avoid race conditions when using library functions.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

#### Risk

The functions flagged by this defect are nonreentrant because their implementations can use global or static variables. When multiple tasks call the function without protection, the function call from one task can interfere with the call from another task. The two invocations of the function can concurrently access the global or static variables and cause unpredictable results.

The calls can also cause more serious security vulnerabilities, such as abnormal termination, denial-of-service attack, and data integrity violations.

#### Fix

To fix this defect, do one of the following:

- Use a reentrant version of the standard library function if it exists.

  For instance, instead of `strerror()`, use `strerror_r()` or `strerror_s()`. For alternatives to functions flagged by this defect, see the documentation for CON33-C.
- Protect the function calls using common critical sections or temporal exclusion.

  See `Critical section details (-critical-section-begin -critical-section-end)` and `Temporally exclusive tasks (-temporal-exclusions-file)`.

  To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon. For an example, see below.

#### Example - Unprotected Call to Standard Library Function from Multiple Tasks

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno); //Noncompliant
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
    end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Specification | |
|---|---|---|
| `Configure multitasking manually` | ☑ | |
| `Tasks (-entry-points)` | `task1`<br>`task2`<br>`task3` | |
| `Critical section details (-critical-section-begin -critical-section-end)` | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

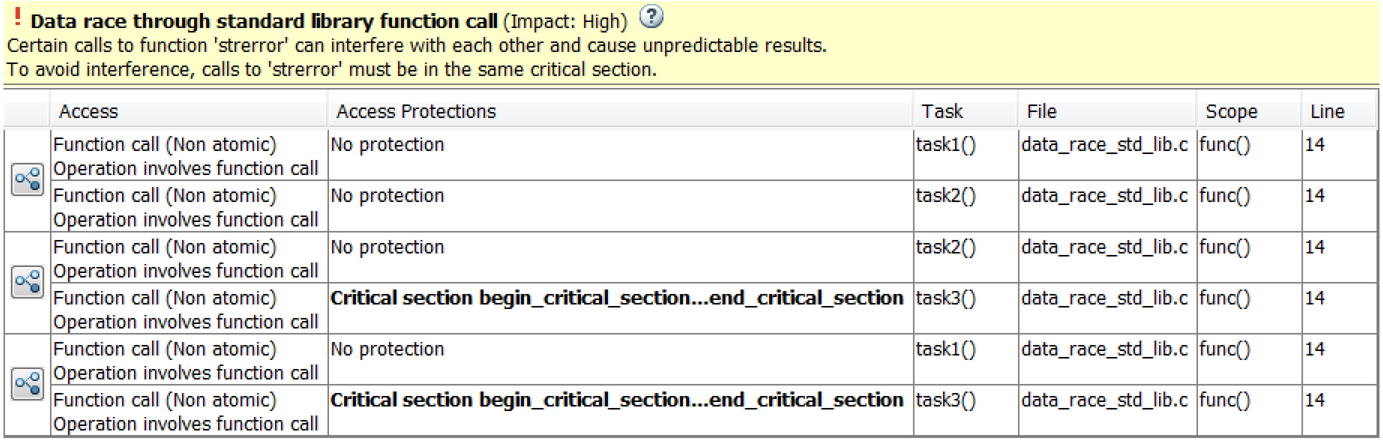
On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2,task3
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

In this example, the tasks, `task1`, `task2` and `task3`, call the function `func`. `func` calls the nonreentrant standard library function, `strerror`.

Though `task3` calls `func` inside a critical section, other tasks do not use the same critical section. Operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

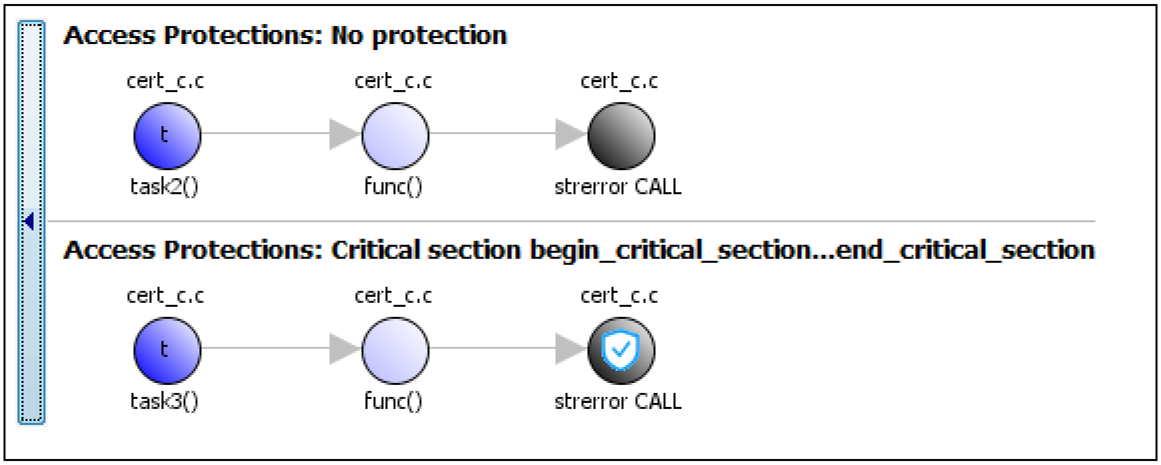
These three tasks are calling a nonreentrant standard library function without common protection. In your result details, you see each pair of conflicting function calls.



! **Data race through standard library function call** (Impact: High)
Certain calls to function 'strerror' can interfere with each other and cause unpredictable results.
To avoid interference, calls to 'strerror' must be in the same critical section.

| | Access | Access Protections | Task | File | Scope | Line |
|---|---|---|---|---|---|---|
| | Function call (Non atomic)<br>Operation involves function call | No protection | task1() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task2() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task2() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | **Critical section begin_critical_section...end_critical_section** | task3() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task1() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | **Critical section begin_critical_section...end_critical_section** | task3() | data_race_std_lib.c | func() | 14 |

If you click the icon, you see the function call sequence starting from the entry point to the standard library function call. You also see that the call starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.


Access Protections: No protection
cert_c.c
t
task2()
cert_c.c
func()
cert_c.c
strerror CALL
Access Protections: Critical section begin_critical_section...end_critical_section
cert_c.c
t
task3()
cert_c.c
func()
cert_c.c
strerror CALL

**Correction — Use Reentrant Version of Standard Library Function**

One possible correction is to use a reentrant version of the standard library function `strerror`. You can use the POSIX version `strerror_r` which has the same functionality but also guarantees thread-safety.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);
enum { BUFFERSIZE = 64 };

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char errmsg[BUFFERSIZE];
    if (strerror_r(errno, errmsg, BUFFERSIZE) != 0) {
      /* Handle error */
    }
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}
```

```
void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

**Correction — Place Function Call in Critical Section**

One possible correction is to place the call to `strerror` in critical section. You can implement the critical section in multiple ways.

For instance, you can place the call to the intermediate function `func` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `func` and therefore the calls to `strerror` from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call `func` between calls to `begin_critical_section` and `end_critical_section`.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    begin_critical_section();
    func(fptr1);
    end_critical_section();
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     begin_critical_section();
     func(fptr2);
     end_critical_section();
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
    end_critical_section();
}
```

#### Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

| Option | Value |
|---|---|
| `Temporally exclusive tasks (-temporal-exclusions-file)` | `task1 task2 task3` |

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

## Check Information

**Group:** 10. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CON33-C

# CERT C++: CON37-C

Do not call signal() in a multithreaded program

## Description

### Rule Definition

*Do not call signal() in a multithreaded program.*

### Polyspace Implementation

The rule checker checks for **Signal call in multithreaded program**.

## Examples

### Signal call in multithreaded program

#### Issue

**Signal call in multithreaded program** occurs when you use the `signal()` function in a program with multiple threads.

#### Risk

According to the C11 standard (Section 7.14.1.1), use of the `signal()` function in a multithreaded program is undefined behavior.

#### Fix

Depending on your intent, use other ways to perform an asynchronous action on a specific thread.

#### Example - Use of `signal()` Function to Terminate Loop in Thread

```
#include <signal.h>
#include <stddef.h>
#include <thread>

volatile sig_atomic_t flag = 0;

void handler(int signum) {
    flag = 1;
}

/* Runs until user sends SIGUSR1 */
int func(int data) {
    while (!flag) {
        /* ... */
    }
    return 0;
}

int main(void) {
    signal(SIGINT, handler); //Noncompliant
    int data;
    //...
    std::thread th1(func, data);

    return 0;
}
```

In this example, the `signal` function is used to terminate a `while` loop in the thread.

#### Correction — Use `atomic_bool` Variable to Terminate Loop

One possible correction is to use an `std::atomic` variable of `bool` type that multiple threads can access. In the corrected example, the `std::atomic<bool>` variable `flag` can be accessed by both the main thread and the child thread `th1`. Before every loop iteration, the child thread checks `flag`. After completing the program, you can modify this variable so that the child thread exits the loop.

```
#include <thread>
#include <atomic>
std::atomic<bool> flag(false);

int func(int data) {
    while (!flag) {
        /* ... */
    }
    return 0;
}

int main(void) {
    int data;
    //...
    std::thread th1(func, data);
    th1.join();
    flag = true;
    return 0;
}
```

## Check Information

**Group:** 10. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

CON37-C

# CERT C++: CON40-C

Do not refer to an atomic variable twice in an expression

## Description

### Rule Definition

*Do not refer to an atomic variable twice in an expression.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Atomic variable accessed twice in an expression**.
- **Atomic load and store sequence not atomic**.

## Examples

### Atomic variable accessed twice in an expression

#### Issue

**Atomic variable accessed twice in an expression** occurs when C atomic types or C++ `std::atomic` class variables appear twice in an expression and there are:

- Two atomic read operations on the variable.
- An atomic read and a distinct atomic write operation on the variable.

The C standard defines certain operations on atomic variables that are thread safe and do not cause data race conditions. Unlike individual operations, a pair of operations on the same atomic variable in an expression is not thread safe.

#### Risk

A thread can modify the atomic variable between the pair of atomic operations, which can result in a data race condition.

#### Fix

Do not reference an atomic variable twice in the same expression.

#### Example - Referencing Atomic Variable Twice in an Expression

To run this example, use these options:

- `-cpp-version cpp11`
- -compiler gnu4.9

```
#include <atomic>
std::atomic<int> n(5);
int compute_sum(void)
{
    return n * (n + 1) / 2; //Noncompliant
}
```

In this example, the global variable `n` is referenced twice in the return statement of `compute_sum()`. The value of `n` can change between the two distinct read operations. `compute_sum()` can return an incorrect value.

#### Correction — Pass Variable as Function Argument

One possible correction is to pass the variable as a function argument `n`. The variable is copied to memory and the read operations on the copy guarantee that `compute_sum()` returns a correct result. If you pass a variable of type `int` instead of type `atomic_int`, the correction is still valid.

```
#include <atomic>
int compute_sum(std::atomic<int> n)
{
    return n * (n + 1) / 2;
}
```

### Atomic load and store sequence not atomic

#### Issue

**Atomic load and store sequence not atomic** occurs when you use these functions to load, and then store an atomic variable.

- C functions:
  - `atomic_load()`
  - `atomic_load_explicit()`
  - `atomic_store()`
  - `atomic_store_explicit()`
- C++ functions:
  - `std::atomic_load()`
  - `std::atomic_load_explicit()`
  - `std::atomic_store()`
  - `std::atomic_store_explicit()`
  - `std::atomic::load()`
  - `std::atomic::store()`

A thread cannot interrupt an atomic load or an atomic store operation on a variable, but a thread can interrupt a store, and then load sequence.

#### Risk

A thread can modify a variable between the load and store operations, resulting in a data race condition.

#### Fix

To read, modify, and store a variable atomically, use a compound assignment operator such as +=, `atomic_compare_exchange()` or `atomic_fetch_*`-family functions.

### Example - Loading Then Storing an Atomic Variable

To run this example, use these options:

- `-cpp-version cpp11`
- -compiler gnu4.9

```
#include <atomic>
#include <stdbool.h>
using namespace std;
static atomic<bool> flag(false);

void init_flag(void)
{
    atomic_init(&flag, false);
}

void toggle_flag(void)
{
    bool temp_flag = atomic_load(&flag);
    temp_flag = !temp_flag;
    atomic_store(&flag, temp_flag); //Noncompliant
}

bool get_flag(void)
{
    return atomic_load(&flag);
}
```

In this example, variable `flag` of type `atomic_bool` is referenced twice inside the `toggle_flag()` function. The function loads the variable, negates its value, then stores the new value back to the variable. If two threads call `toggle_flag()`, the second thread can access `flag` between the load and store operations of the first thread. `flag` can end up in an incorrect state.

#### Correction — Use Compound Assignment to Modify Variable

One possible correction is to use the function `atomic_compare_exchange_weak` to perform a safe and atomic compare-and-exchange. When you use this function, the changes to `flag` are visible to other threads, and the expected result is stored in `flag`.

```
#include <atomic>
#include <stdbool.h>
using namespace std;
static atomic<bool> flag(false);


void toggle_flag(void)
{
  bool old_flag = atomic_load(&flag);
  bool new_flag;
  do {
    new_flag = !old_flag;
  } while (!atomic_compare_exchange_weak(&flag, &old_flag, new_flag));

}

bool get_flag(void)
{
    return atomic_load(&flag);
}
```

## Check Information

**Group:** 10. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CON40-C

# CERT C++: CON41-C

Wrap functions that can fail spuriously in a loop

## Description

### Rule Definition

*Wrap functions that can fail spuriously in a loop.*

### Polyspace Implementation

The rule checker checks for **Function that can spuriously fail not wrapped in loop**.

## Examples

### Function that can spuriously fail not wrapped in loop

#### Issue

**Function that can spuriously fail not wrapped in loop** occurs when the following atomic compare and exchange functions that can fail spuriously are called from outside a loop.

- C atomic functions:
  - `atomic_compare_exchange_weak()`
  - `atomic_compare_exchange_weak_explicit()`
- C++ atomic functions:
  - `std::atomic<T>::compare_exchange_weak(T* expected, T desired)`
  - `std::atomic<T>::compare_exchange_weak_explicit(T* expected, T desired, std::memory_order succ, std::memory_order fail)`
  - `std::atomic_compare_exchange_weak(std::atomic<T>* obj, T* expected, T desired)`
  - `std::atomic_compare_exchange_weak_explicit(volatile std::atomic<T>* obj, T* expected, T desired, std::memory_order succ, std::memory_order fail)`

The functions compare the memory contents of the object representations pointed to by `obj` and `expected`. The comparison can spuriously return false even if the memory contents are equal. This spurious failure makes the functions faster on some platforms.

#### Risk

An atomic compare and exchange function that spuriously fails can cause unexpected results and unexpected control flow.

#### Fix

Wrap atomic compare and exchange functions that can spuriously fail in a loop. The loop checks the failure condition after a possible spurious failure.

### Example — `atomic_compare_exchange_weak()` Not Wrapped in Loop

```
#include <atomic>
#include <stdbool.h>
using namespace std;
static atomic<bool> flag(false);
void toggle_flag(void)
{
    bool old_flag = atomic_load(&flag);
    bool new_flag;
    if (!atomic_compare_exchange_weak(&flag, &old_flag, new_flag)){//Noncompliant
        new_flag = !old_flag;
    }

}

bool get_flag(void)
{
    return atomic_load(&flag);
}
```

In this example, the function`toggle_flag` uses `atomic_compare_exchange_weak()` to compare `flag` and `old_flag`. If the variables are identical, `flag` is toggles to `new_flag`. When `atomic_compare_exchange_weak()` fails spuriously, the flag is toggled unnecessarily.

### Correction — Wrap `atomic_compare_exchange_weak()` in a `do-while` Loop

One possible correction is to wrap the call to `atomic_compare_exchange_weak()` in a `while` loop. The loop checks the failure condition after a possible spurious failure.

```
#include <atomic>
#include <stdbool.h>
using namespace std;
static atomic<bool> flag(false);


void toggle_flag(void)
{
  bool old_flag = atomic_load(&flag);
  bool new_flag;
  do {
    new_flag = !old_flag;
  } while (!atomic_compare_exchange_weak(&flag, &old_flag, new_flag));

}

bool get_flag(void)
{
    return atomic_load(&flag);
}
```

## Check Information

**Group:** 10. Concurrency (CON)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

CON41-C

# CERT C++: CON43-C

Do not allow data races in multithreaded code

## Description

### Rule Definition

*Do not allow data races in multithreaded code.*

### Polyspace Implementation

The rule checker checks for **Data race**.

### Extend Checker

For the rule checker to detect issues, your code must use one of the concurrency primitives recognized by Polyspace for thread creation and shared variable protection. Otherwise, you must explicitly specify tasks, interrupts, critical sections, and other multitasking options in your project configuration. See “Multitasking”.

You can also extend this checker in the following ways:

- Map your multithreading functions to their known POSIX equivalents. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.
- Detect all data races including ones involving atomic operations.

  Polyspace assumes that certain operations are atomic and excludes them from data race checks. See “Define Atomic Operations in Multitasking Code”. These assumptions might not apply to your environment. To extend the data race checkers to include these operations, use the option `-detect-atomic-data-race`. See “Extend Data Race Checkers to Atomic Operations”.

## Examples

### Data race

#### Issue

Data race occurs when:

- Multiple tasks perform unprotected operations on a shared variable.
- At least one task performs a write operation.
- At least one operation is nonatomic. To detect data race on both atomic and nonatomic operations, use the options `-detect-atomic-data-race`. See “Extend Data Race Checkers to Atomic Operations”.

  See “Define Atomic Operations in Multitasking Code”.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

### Risk

Data race can result in unpredictable values of the shared variable because you do not control the order of the operations in different tasks.

Data races between two write operations are more serious than data races between a write and read operation. Two write operations can interfere with each other and result in indeterminate values. To identify write-write conflicts, use the filters on the **Detail** column of the **Results List** pane. For these conflicts, the **Detail** column shows the additional line:

```
 Variable value may be altered by write-write concurrent access.
```

See “Filter and Group Results in Polyspace Desktop User Interface” or “Filter and Sort Results in Polyspace Access Web Interface”.

### Fix

To fix this defect, protect the operations on the shared variable using critical sections, temporal exclusion or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon. For an example, see below.

### Example - Unprotected Operation on Global Variable from Multiple Tasks

```
int var; //Noncompliant
void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
    var++;
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Specification | |
|---|---|---|
| `Configure multitasking manually` | ☑ | |
| `Tasks (-entry-points)` | `task1`<br>`task2`<br>`task3` | |
| `Critical section details (-critical-section-begin -critical-section-end)` | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2,task3
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

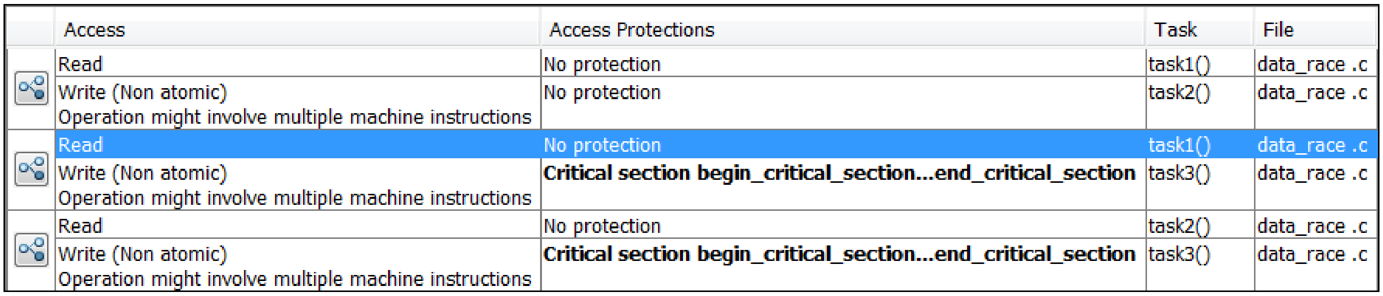
In this example, the tasks `task1`, `task2`, and `task3` call the function `increment`. `increment` contains the operation `var++` that can involve multiple machine instructions including:

- Reading `var`.
- Writing an increased value to `var`.

These machine instructions, when executed from `task1` and `task2`, can occur concurrently in an unpredictable sequence. For example, reading `var` from `task1` can occur either before or after writing to `var` from `task2`. Therefore the value of `var` can be unpredictable.

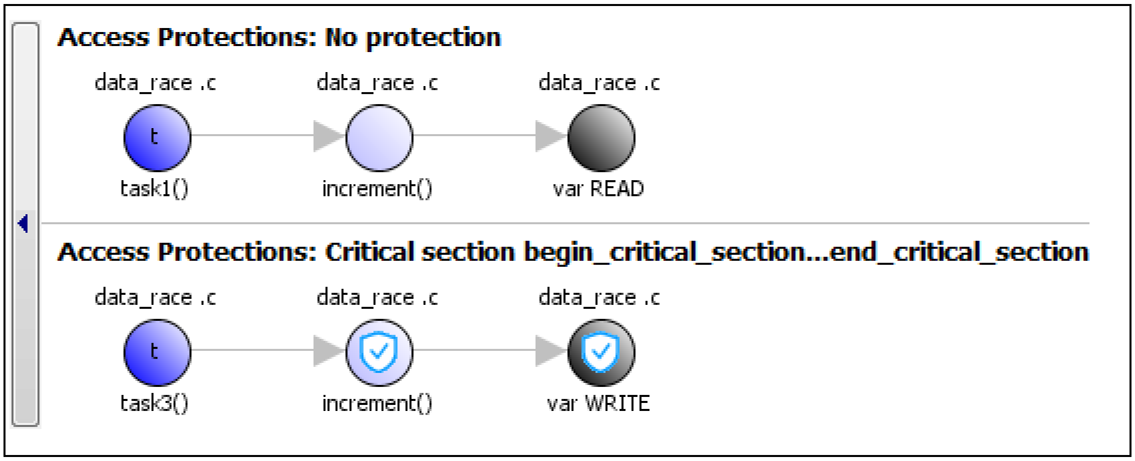
Though `task3` calls `increment` inside a critical section, other tasks do not use the same critical section. The operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

Therefore, the three tasks are operating on a shared variable without common protection. In your result details, you see each pair of conflicting function calls.



| Access | Access Protections | Task | File |
|---|---|---|---|
| Read | No protection | task1() | data_race .c |
| Write (Non atomic)<br>Operation might involve multiple machine instructions | No protection | task2() | data_race .c |
| Read | No protection | task1() | data_race .c |
| Write (Non atomic)<br>Operation might involve multiple machine instructions | **Critical section begin_critical_section...end_critical_section** | task3() | data_race .c |
| Read | No protection | task2() | data_race .c |
| Write (Non atomic)<br>Operation might involve multiple machine instructions | **Critical section begin_critical_section...end_critical_section** | task3() | data_race .c |

If you click the icon, you see the function call sequence starting from the entry point to the read or write operation. You also see that the operation starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.


**Access Protections: No protection**

**Access Protections: Critical section begin_critical_section...end_critical_section**

**Correction — Place Operation in Critical Section**

One possible correction is to place the operation in critical section. You can implement the critical section in multiple ways. For instance:

- You can place `var++` in a critical section. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The operation `var++` from the three tasks cannot interfere with each other.

  To implement the critical section, in the function `increment`, place the operation `var++` between calls to `begin_critical_section` and `end_critical_section`.

  ```
  int var;

  void begin_critical_section(void);
  void end_critical_section(void);

  void increment(void) {
        begin_critical_section();
        var++;
        end_critical_section();
  }

  void task1(void)  {
        increment();
  }

  void task2(void)  {
        increment();
  }

  void task3(void)  {
        increment();
  }
  ```
- You can place the call to `increment` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `increment` from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call `increment` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      var++;
}

void task1(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task2(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

#### Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

| Option | Value |
|---|---|
| Temporally exclusive tasks (-temporal-exclusions-file) | task1 task2 task3 |

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

#### Example - Unprotected Operation in Threads Created with `pthread_create`

```
#include <pthread.h>
```

```
pthread_mutex_t count_mutex;
long long count; //Noncompliant


void* increment_count(void* args)
{
    count = count + 1;
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    c = count;
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

In this example, Bug Finder detects the creation of separate threads with `pthread_create`. The **Data race** defect is raised because the operation `count = count + 1` in the thread with id `thread_increment` conflicts with the operation `c = count` in the thread with id `thread_get`. The variable `count` is accessed in multiple threads without a common protection.

The two conflicting operations are nonatomic. The operation `c = count` is nonatomic on 32-bit targets. See “Define Atomic Operations in Multitasking Code”.

#### Correction — Protect Operations with `pthread_mutex_lock` and `pthread_mutex_unlock` Pair

To prevent concurrent access on the variable `count`, protect operations on `count` with a critical section. Use the functions `pthread_mutex_lock` and `pthread_mutex_unlock` to implement the critical section.

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}
```

```
void* set_count(void *args)
{
    long long c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

## Check Information

**Group:** 10. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CON43-C

# CERT C++: CON50-CPP

Do not destroy a mutex while it is locked

## Description

### Rule Definition

*Do not destroy a mutex while it is locked.*

### Polyspace Implementation

The rule checker checks for **Destruction of locked mutex**.

## Examples

### Destruction of locked mutex

#### Issue

**Destruction of locked mutex** occurs when a task destroys a mutex after it is locked (and before it is unlocked). The locking and destruction can happen in the same task or different tasks.

#### Risk

A mutex is locked to protect shared variables from concurrent access. If a mutex is destroyed in the locked state, the protection does not apply.

#### Fix

To fix this defect, destroy the mutex only after you unlock it. It is a good design practice to:

- Initialize a mutex *before* creating the threads where you use the mutex.
- Destroy a mutex *after* joining the threads that you created.

On the **Result Details** pane, you see two events, the locking and destruction of the mutex, and the tasks that initiated the events. To navigate to the corresponding line in your source code, click the event.

#### Example - Locking and Destruction in Different Tasks

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
  pthread_mutex_unlock (&lock3);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);
  pthread_mutex_destroy (&lock3); //Noncompliant
  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

In this example, after task `t0` locks the mutex `lock3`, task `t1` can destroy it. The destruction occurs if the following events happen in sequence:

1. `t0` acquires `lock3`.
2. `t0` releases `lock2`.
3. `t0` releases `lock1`.
4. `t1` acquires the lock `lock1` released by `t0`.
5. `t1` acquires the lock `lock2` released by `t0`.
6. `t1` destroys `lock3`.

For simplicity, this example uses a mix of automatic and manual concurrency detection. The tasks `t0` and `t1` are manually specified as entry points by using the option `Tasks (-entry-points)`.The critical sections are implemented through primitives `pthread_mutex_lock` and `pthread_mutex_unlock` that the software detects automatically. In practice, for entry point specification (thread creation), you will use primitives such as `pthread_create`. The next example shows how the defect can appear when you use `pthread_create`.

**Correction — Place Lock-Unlock Pair Together in Same Critical Section as Destruction**

The locking and destruction of `lock3` occurs inside the critical section imposed by `lock1` and `lock2`, but the unlocking occurs outside. One possible correction is to place the lock-unlock pair in the same critical section as the destruction of the mutex. Use one of these critical sections:

- Critical section imposed by `lock1` alone.
- Critical section imposed by `lock1` and `lock2`.

In this corrected code, the lock-unlock pair and the destruction is placed in the critical section imposed by `lock1` and `lock2`. When `t0` acquires `lock1` and `lock2`, `t1` has to wait for their release before it executes the instruction `pthread_mutex_destroy (&lock3);`. Therefore, `t1` cannot destroy mutex `lock3` in the locked state.

```
#include <pthread.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;
pthread_mutex_t lock3;

void t0 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);

  pthread_mutex_lock (&lock3);
  pthread_mutex_unlock (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}

void t1 (void) {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock2);

  pthread_mutex_destroy (&lock3);

  pthread_mutex_unlock (&lock2);
  pthread_mutex_unlock (&lock1);
}
```

**Example - Locking and Destruction in Start Routine of Thread**

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_exit((void*) 0);
}

void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_destroy(&lock);//Noncompliant
    pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Thread that initializes mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use mutex for atomic operation*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   pthread_exit(NULL);
}
```

In this example, four threads are created. The threads are assigned different actions.

- The first thread `callThd[0]` initializes the mutex `lock`.
- The second and third threads, `callThd[1]` and `callThd[2]`, perform an atomic operation protected by the mutex `lock`.
- The fourth thread `callThd[3]` destroys the mutex `lock`.

The threads can interrupt each other. Therefore, immediately after the second or third thread locks the mutex, the fourth thread can destroy it.

#### Correction — Initialize and Destroy Mutex Outside Start Routine

One possible correction is to initialize and destroy the mutex in the `main` function outside the start routine of the threads. The threads perform only the atomic operation. You need two fewer threads because the mutex initialization and destruction threads are not required.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 2
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
void atomic_operation(void);

void *do_work(void *arg) {
   pthread_mutex_lock (&lock);
   atomic_operation();
   pthread_mutex_unlock (&lock);
   pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize mutex */
   pthread_mutex_init(&lock, NULL);

   for(i=0; i<NUMTHREADS; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy mutex */
   pthread_mutex_destroy(&lock);

   pthread_exit(NULL);
}
```

#### Correction — Use A Second Mutex To Protect Lock-Unlock Pair and Destruction

Another possible correction is to use a second mutex and protect the lock-unlock pair from the destruction. This corrected code uses the mutex `lock2` to achieve this protection. The second mutex is initialized in the `main` function outside the start routine of the threads.

```
#include <pthread.h>

/* Define globally accessible variables and a mutex */
#define NUMTHREADS 4
pthread_t callThd[NUMTHREADS];
pthread_mutex_t lock;
pthread_mutex_t lock2;
void atomic_operation(void);

void *do_create(void *arg) {
    /* Creation thread */
    pthread_mutex_init(&lock, NULL);
    pthread_exit((void*) 0);
}

void *do_work(void *arg) {
    /* Worker thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_lock (&lock);
    atomic_operation();
    pthread_mutex_unlock (&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}
```

```
void *do_destroy(void *arg) {
    /* Destruction thread */
    pthread_mutex_lock (&lock2);
    pthread_mutex_destroy(&lock);
    pthread_mutex_unlock (&lock2);
    pthread_exit((void*) 0);
}


int main (int argc, char *argv[]) {
   int i;
   void *status;
   pthread_attr_t attr;


   /* Create threads */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   /* Initialize second mutex */
   pthread_mutex_init(&lock2, NULL);

   /* Thread that initializes first mutex */
   pthread_create(&callThd[0], &attr, do_create, NULL);

   /* Threads that use first mutex for atomic operation */
   /* The threads use second mutex to protect first from destruction in locked state*/
   for(i=0; i<NUMTHREADS-1; i++) {
      pthread_create(&callThd[i], &attr, do_work, (void *)i);
   }

   /* Thread that destroys first mutex */
   /* The thread uses the second mutex to prevent destruction of locked mutex */
   pthread_create(&callThd[NUMTHREADS -1], &attr, do_destroy, NULL);


   pthread_attr_destroy(&attr);

   /* Join threads */
   for(i=0; i<NUMTHREADS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* Destroy second mutex */
   pthread_mutex_destroy(&lock2);

   pthread_exit(NULL);
}
```

## Check Information

**Group:** 10. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**

“Check for and Review Coding Standard Violations”

**External Websites**

CON50-CPP

# CERT C++: CON52-CPP

Prevent data races when accessing bit-fields from multiple threads

## Description

### Rule Definition

*Prevent data races when accessing bit-fields from multiple threads.*

### Polyspace Implementation

The rule checker checks for **Data race on adjacent bit fields**.

### Extend Checker

For the rule checker to detect issues, your code must use one of the concurrency primitives recognized by Polyspace for thread creation and shared variable protection. Otherwise, you must explicitly specify tasks, interrupts, critical sections, and other multitasking options in your project configuration. See “Multitasking”.

## Examples

### Data race on adjacent bit fields

#### Issue

This defect occurs when:

- Multiple tasks perform unprotected operations on bit fields that are part of the same structure.

  For instance, a task operates on field `errorFlag1` and another task on field `errorFlag2` in a variable of this type:

  ```
  struct errorFlags {
     unsigned int errorFlag1 : 1;
     unsigned int errorFlag2 : 1;
     ...
  }
  ```

  Suppose that the operations are not atomic with respect to each other. In other words, you have not implemented protection mechanisms to ensure that one operation is completed before another operation begins.
- At least one of the unprotected operations is a write operation.

To find this defect, before analysis, you must specify the multitasking options. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

#### Risk

Adjacent bit fields that are part of the same structure might be stored in one byte in the same memory location. Read or write operations on all variables including bit fields occur one byte or word at a time. To modify only specific bits in a byte, steps similar to these steps occur in sequence:

1. The byte is loaded into RAM.
2. A mask is created so that only specific bits are modified to the intended value and the remaining bits remain unchanged.
3. A bitwise OR operation is performed between the copy of the byte in RAM and the mask.
4. The byte with specific bits modified is copied back from RAM.

When you access two different bit fields, these four steps have to be performed for each bit field. If the accesses are not protected, all four steps for one bit field might not be completed before the four steps for the other bit field begin. As a result, the modification of one bit field might undo the modification of an adjacent bit field. For instance, in the preceding example, the modification of `errorFlag1` and `errorFlag2` can occur in the following sequence. Steps 1,2 and 5 relate to modification of `errorFlag1` and while steps 3,4 and 6 relate to that of `errorFlag2`.

1. The byte with both `errorFlag1` and `errorFlag2` unmodified is copied into RAM, for purposes of modifying `errorFlag1`.
2. A mask that modifies only `errorFlag1` is bitwise OR-ed with this copy.
3. The byte containing both `errorFlag1` and `errorFlag2` unmodified is copied into RAM a second time, for purposes of modifying `errorFlag2`.
4. A mask that modifies only `errorFlag2` is bitwise OR-ed with this second copy.
5. The version with `errorFlag1` modified is copied back. This version has `errorFlag2` unmodified.
6. The version with `errorFlag2` modified is copied back. This version has `errorFlag1` unmodified and overwrites the previous modification.

#### Fix

To fix this defect, protect the operations on bit fields that are part of the same structure by using critical sections, temporal exclusion, or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon.

#### Example - Unprotected Operation on Global Variable from Multiple Tasks

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12; //Noncompliant

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

In this example, `task1` and `task2` access different bit fields `IOFlag` and `SetupFlag`, which belong to the same structured variable `InterruptConfigbitsProc12`.

To emulate multitasking behavior, specify the options listed in this table.

| Option | Specification |
|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ |
| **Tasks** on page 2-121 | `task1`<br>`task2` |

At the command-line, use:

```
 polyspace-bug-finder
   -entry-points task1,task2
```

**Correction – Use Critical Sections**

One possible correction is to wrap the bit field access in a critical section. A critical section lies between a call to a lock function and an unlock function. In this correction, the critical section lies between the calls to functions `begin_critical_section` and `end_critical_section`.

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void begin_critical_section(void);
void end_critical_section(void);

void task1 (void) {
    begin_critical_section();
    InterruptConfigbitsProc12.IOFlag = 0;
    end_critical_section();
}

void task2 (void) {
    begin_critical_section();
    InterruptConfigbitsProc12.SetupFlag = 0;
    end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify options listed in this table.

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ | |
| **Tasks** on page 2-121 | `task1`<br>`task2` | |
| **Critical section details** on page 2-133 | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

At the command-line, use:

```
 polyspace-bug-finder
   -entry-points task1,task2
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

#### Correction – Avoid Bit Fields

If you do not have memory constraints, use the `char` data type instead of bit fields. The `char` variables in a structure occupy at least one byte and do not have the thread-safety issues that come from bit manipulations in a byte-sized operation. Data races do not result from unprotected operations on different `char` variables that are part of the same structure.

```
typedef struct
{
   unsigned char IOFlag;
   unsigned char InterruptFlag;
   unsigned char Register1Flag;
   unsigned char SignFlag;
   unsigned char SetupFlag;
   unsigned char Register2Flag;
   unsigned char ProcessorFlag;
   unsigned char GeneralFlag;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

Though the checker does not flag this correction, do not use this correction for C99 or earlier. Only from C11 and later does the C Standard mandate that distinct `char` variables cannot be accessed using the same word.

#### Correction – Insert Bit Field of Size 0

You can enter a non-bit field member or an unnamed bit field member of size 0 between two adjacent bit fields that might be accessed concurrently. A non-bit field member or size 0 bit field member ensures that the subsequent bit field starts from a new memory location. In this corrected example, the size 0 bit field member ensures that `IOFlag` and `SetupFlag` are stored in distinct memory locations.

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int : 0;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

## Check Information

**Group:** 10. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CON52-CPP

# CERT C++: CON53-CPP

Avoid deadlock by locking in a predefined order

## Description

### Rule Definition

*Avoid deadlock by locking in a predefined order.*

### Polyspace Implementation

The rule checker checks for **Deadlock**.

### Extend Checker

For the rule checker to detect issues, your code must use one of the concurrency primitives recognized by Polyspace for thread creation and shared variable protection. Otherwise, you must explicitly specify tasks, interrupts, critical sections, and other multitasking options in your project configuration. See “Multitasking”.

You can also extend this checker by mapping your multithreading functions to their known POSIX equivalents. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.

## Examples

### Deadlock

#### Issue

**Deadlock** occurs when multiple tasks are stuck in their critical sections (CS) because:

- Each CS waits for another CS to end.
- The critical sections (CS) form a closed cycle. For example:
  - CS #1 waits for CS #2 to end, and CS #2 waits for CS #1 to end.
  - CS #1 waits for CS #2 to end, CS #2 waits for CS #3 to end and CS #3 waits for CS #1 to end.

Polyspace expects critical sections of code to follow a specific format. A critical section lies between a call to a lock function and a call to an unlock function. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function. Both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

#### Risk

Each task waits for a critical section in another task to end and is unable to proceed. The program can freeze indefinitely.

#### Fix

The fix depends on the root cause of the defect. You can try to break the cyclic order between the tasks in one of these ways:

- Write down all critical sections involved in the deadlock in a certain sequence. Whenever you call the lock functions of the critical sections within a task, respect the order in that sequence. See an example below.
- If one of the critical sections involved in a deadlock occurs in an interrupt, try to disable all interrupts during critical sections in all tasks. See `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Reviewing this defect is an opportunity to check if all operations in your critical section are really meant to be executed as an atomic block. It is a good practice to keep critical sections at a bare minimum.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example - Deadlock with Two Tasks

```
void task1(void);
void task2(void);

int var;
void perform_task_cycle(void) {
 var++;
}

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2(); //Noncompliant
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_2();
    begin_critical_section_1();
    perform_task_cycle();
    end_critical_section_1();
    end_critical_section_2();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** | ☑ | |
| **Entry points** | task1<br>task2 | |
| **Critical section details** | **Starting routine** | **Ending routine** |
| | begin_critical_section_1 | end_critical_section_1 |
| | begin_critical_section_2 | end_critical_section_2 |

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. task1 calls begin_critical_section_1.
2. task2 calls begin_critical_section_2.
3. task1 reaches the instruction begin_critical_section_2();. Since task2 has already called begin_critical_section_2, task1 waits for task2 to call end_critical_section_2.
4. task2 reaches the instruction begin_critical_section_1();. Since task1 has already called begin_critical_section_1, task2 waits for task1 to call end_critical_section_1.

#### Correction-Follow Same Locking Sequence in Both Tasks

One possible correction is to follow the same sequence of calls to lock and unlock functions in both task1 and task2.

```
void task1(void);
void task2(void);
void perform_task_cycle(void);

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}
```

**Example - Deadlock with More Than Two Tasks**

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);


void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2(); //Noncompliant
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock3();
    lock1();
    performTaskCycle();
    unlock1();
    unlock3();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** | ☑ | |
| **Entry points** | task1<br>task2<br>task3 | |
| **Critical section details** | **Starting routine** | **Ending routine** |
| | lock1 | unlock1 |
| | lock2 | unlock2 |
| | lock3 | unlock3 |

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `lock1`.
2. `task2` calls `lock2`.
3. `task3` calls `lock3`.
4. `task1` reaches the instruction `lock2();`. Since `task2` has already called `lock2`, `task1` waits for call to `unlock2`.
5. `task2` reaches the instruction `lock3();`. Since `task3` has already called `lock3`, `task2` waits for call to `unlock3`.
6. `task3` reaches the instruction `lock1();`. Since `task1` has already called `lock1`, `task3` waits for call to `unlock1`.

#### Correction — Break Cyclic Order

To break the cyclic order between critical sections, note every lock function in your code in a certain sequence, for example:

1. `lock1`
2. `lock2`
3. `lock3`

If you use more than one lock function in a task, use them in the order in which they appear in the sequence. For example, you can use `lock1` followed by `lock2` but not `lock2` followed by `lock1`.

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);

void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock1();
    lock3();
    performTaskCycle();
    unlock3();
    unlock1();
 }
}
```

## Check Information

**Group:** 10. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

CON53-CPP

# CERT C++: CON54-CPP

Wrap functions that can spuriously wake up in a loop

## Description

### Rule Definition

*Wrap functions that can spuriously wake up in a loop.*

### Polyspace Implementation

The rule checker checks for **Function that can spuriously wake up not wrapped in loop**.

## Examples

### Function that can spuriously wake up not wrapped in loop

#### Issue

**Function that can spuriously wake up not wrapped in loop** occurs when the following wait-on-condition functions are called from outside a loop:

- C functions:
  - `cnd_wait()`
  - `cnd_timedwait()`
- POSIX functions:
  - `pthread_cond_wait()`
  - `pthread_cond_timedwait()`
- C++ `std::condition_variable` and `std::condition_variable_any` class member functions:
  - `wait()`
  - `wait_until()`
  - `wait_for()`

Wait-on-condition functions pause the execution of the calling thread when a specified condition is met. The thread wakes up and resumes once another thread notifies it with `cnd_broadcast()` or an equivalent function. The wake-up notification can be spurious or malicious.

#### Risk

If a thread receives a spurious wake-up notification and the condition of the wait-on-condition function is not checked, the thread can wake up prematurely. The wake-up can cause unexpected control flow, indefinite blocking of other threads, or denial of service.

#### Fix

Wrap wait-on-condition functions that can wake up spuriously in a loop. The loop checks the wake-up condition after a possible spurious wake-up notification.

**Example - `std::condition_variable::wait` Not Wrapped in Loop**

```
#include <stdio.h>
#include <stddef.h>
#include <thread>
#include <mutex>

#define THRESHOLD 100

std::mutex myMutex;
std::condition_variable cv;

void func(int input)
{
    std::unique_lock<std::mutex> lk(myMutex);
    // test condition to pause thread
    if (input > THRESHOLD) {
    //pause current thread
        cv.wait(lk);//Noncompliant
    }
}
```

In this example, the thread uses `std::condition_variable::wait` to pause execution when `input` is greater than `THRESHOLD`. The paused thread can resume if another thread uses `std::condition_variable::notify_all`, which notifies all the threads. This notification causes the thread to wake up even if the pause condition is still true.

**Correction — Wrap `std::condition_variable::wait` in a `while` Loop Explicitly**

One possible correction is to wrap the call to `std::condition_variable::wait` in a `while` loop. The loop checks the pause condition after the thread receives a possible spurious wake-up notification.

```
#include <stdio.h>
#include <stddef.h>
#include <thread>
#include <mutex>

#define THRESHOLD 100

std::mutex myMutex;
std::condition_variable cv;

void func(int input)
{
    std::unique_lock<std::mutex> lk(myMutex);
    // test condition to pause thread
    while (input > THRESHOLD) {
    //pause current thread
        cv.wait(lk);
    }
}
```

**Correction — Wrap `std::condition_variable::wait` in a Loop Implicitly**

The `std::condition_variable::wait` function has an overload that accepts a lambda function as a second argument. The predicate of the Lambda function indicates when it is safe to stop waiting and proceed with the code execution. This overload of the `std::condition_variable::wait` function behaves as if it is implicitly wrapped in a loop. In this code, the function`std::condition_variable::wait` is invoked by using a Lambda function. Here, unwanted waking of the thread is prevented because the thread wakes up when the predicate of the Lambda function is true.

```
#include <stdio.h>
#include <stdio.h>
#include <stddef.h>
#include <thread>
#include <mutex>
#define THRESHOLD 100

std::mutex myMutex;
std::condition_variable cv;

void func(int input)
{
    std::unique_lock<std::mutex> lk(myMutex);
    cv.wait(lk,[&input]{ return !(input>THRESHOLD); });

}
```

## Check Information

**Group:** 10. Concurrency (CON)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CON54-CPP

# CERT C++: ENV30-C

Do not modify the object referenced by the return value of certain functions

## Description

### Rule Definition

*Do not modify the object referenced by the return value of certain functions.*

### Polyspace Implementation

The rule checker checks for **Modification of internal buffer returned from nonreentrant standard function**.

## Examples

### Modification of internal buffer returned from nonreentrant standard function

**Issue**

**Modification of internal buffer returned from nonreentrant standard function** occurs when the following happens:

- A nonreentrant standard function returns a pointer.
- You attempt to write to the memory location that the pointer points to.

Nonreentrant standard functions that return a non `const`-qualified pointer to an internal buffer include `getenv`, `getlogin`, `crypt`, `setlocale`, `localeconv`, `strerror` and others.

**Risk**

Modifying the internal buffer that a nonreentrant standard function returns can cause the following issues:

- It is possible that the modification does not succeed or alters other internal data.

  For instance, `getenv` returns a pointer to an environment variable value. If you modify this value, you alter the environment of the process and corrupt other internal data.
- Even if the modification succeeds, it is possible that a subsequent call to the same standard function does not return your modified value.

  For instance, you modify the environment variable value that `getenv` returns. If another process, thread, or signal handler calls `setenv`, the modified value is overwritten. Therefore, a subsequent call to `getenv` does not return your modified value.

**Fix**

Avoid modifying the internal buffer using the pointer returned from the function.

**Example - Modification of `getenv` Return Value**

```
#include <stdlib.h>
#include <string.h>
```

```
void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        strncpy(env, "C", 1); //Noncompliant
        printstr(env);
    }
}
```

In this example, the first argument of `strncpy` is the return value from a nonreentrant standard function `getenv`. The behavior can be undefined because `strncpy` modifies this argument.

#### Correction - Copy Return Value of getenv and Modify Copy

One possible solution is to copy the return value of `getenv` and pass the copy to the `strncpy` function.

```
#include <stdlib.h>
#include <string.h>
enum {
    SIZE20 = 20
};

void printstr(const char*);

void func() {
    char* env = getenv("LANGUAGE");
    if (env != NULL) {
        char env_cp[SIZE20];
        strncpy(env_cp, env, SIZE20);
        strncpy(env_cp, "C", 1);
        printstr(env_cp);
    }
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ENV30-C

# CERT C++: ENV31-C

Do not rely on an environment pointer following an operation that may invalidate it

## Description

### Rule Definition

*Do not rely on an environment pointer following an operation that may invalidate it.*

### Polyspace Implementation

The rule checker checks for **Environment pointer invalidated by previous operation**.

## Examples

### Environment pointer invalidated by previous operation

#### Issue

**Environment pointer invalidated by previous operation** occurs when you use the third argument of *main()* in a hosted environment to access the environment after an operation modifies the environment. In a hosted environment, many C implementations support the nonstandard syntax:

```
main (int argc, char *argv[], char *envp[])
```

A call to a `setenv` or `putenv` family function modifies the environment pointed to by `*envp`.

#### Risk

When you modify the environment through a call to a `setenv` or `putenv` family function, the environment memory can potentially be reallocated. The hosted environment pointer is not updated and might point to an incorrect location. A call to this pointer can return unexpected results or cause an abnormal program termination.

#### Fix

Do not use the hosted environment pointer. Instead, use global external variable `environ` in Linux, `_environ` or `_wenviron` in Windows, or their equivalent. When you modify the environment, these variables are updated.

#### Example - Access Environment Through Pointer envp

```
#include <stdio.h>
#include <stdlib.h>

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

/* envp is from main function */
int func(char **envp)
{
    /* Call to setenv may cause environment
     *memory to be reallocated
     */
    if (setenv(("MY_NEW_VAR"),("new_value"),1) != 0)
    {
        /* Handle error */
        return -1;
    }
    /* envp not updated after call to setenv, and may
     *point to incorrect location.
     **/
    if (envp != ((void *)0)) { //Noncompliant
        use_envp(envp);
/* No defect on second access to
*envp because defect already raised */
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func(envp);
    }
}
```

In this example, `envp` is accessed inside `func()` after a call to `setenv` that can reallocate the environment memory. `envp` can point to an incorrect location because it is not updated after `setenv` modifies the environment. No defect is raised when `use_envp()` is called because the defect is already raised on the previous line of code.

**Correction — Use Global External Variable `environ`**

One possible correction is to access the environment by using a variable that is always updated after a call to `setenv`. For instance, in the following code, the pointer `envp` is still available from `main()`, but the environment is accessed in `func()` through the global external variable `environ`.

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

int func(void)
{
    if (setenv(("MY_NEW_VAR"), ("new_value"),1) != 0) {
        /* Handle error */
        return -1;
    }
  /* Use global external variable environ
   *which is always updated after a call to setenv */

    if (environ != NULL) {
        use_envp(environ);
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
```

```
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func();
    }
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

ENV31-C

# CERT C++: ENV32-C

All exit handlers must return normally

## Description

### Rule Definition

*All exit handlers must return normally.*

### Polyspace Implementation

The rule checker checks for **Abnormal termination of exit handler**.

## Examples

### Abnormal termination of exit handler

#### Issue

**Abnormal termination of exit handler** looks for registered exit handlers. Exit handlers are registered with specific functions such as `atexit`, (WinAPI) `_onexit`, or `at_quick_exit()`. If the exit handler calls a function that interrupts the program's expected termination sequence, Polyspace raises a defect. Some functions that can cause abnormal exits are `exit`, `abort`, `longjmp`, or (WinAPI) `_onexit`.

#### Risk

If your exit handler terminates your program, you can have undefined behavior. Abnormal program termination means other exit handlers are not invoked. These additional exit handlers may do additional clean up or other required termination steps.

#### Fix

In inside exit handlers, remove calls to functions that prevent the exit handler from terminating normally.

#### Example - Exit Handler With Call to exit

```
#include <stdlib.h>

volatile int some_condition = 1;
void demo_exit1(void)
{
    /* ... Cleanup code ... */
    return;
}
void exitabnormalhandler(void)
{
    if (some_condition)
    {
        /* Clean up */
        exit(0);  //Noncompliant
    }
    return;
}

int demo_install_exitabnormalhandler(void)
{

    if (atexit(demo_exit1) != 0) /* demo_exit1() performs additional cleanup */
    {
        /* Handle error */
    }
    if (atexit(exitabnormalhandler) != 0)
    {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

In this example, `demo_install_exitabnormalhandler` registers two exit handlers, `demo_exit1` and `exitabnormalhandler`. Exit handlers are invoked in the reverse order of which they are registered. When the program ends, `exitabnormalhandler` runs, then `demo_exit1`. However, `exitabnormalhandler` calls `exit` interrupting the program exit process. Having this `exit` inside an exit handler causes undefined behavior because the program is not finished cleaning up safely.

#### Correction — Remove exit from Exit Handler

One possible correction is to let your exit handlers terminate normally. For this example, `exit` is removed from `exitabnormalhandler`, allowing the exit termination process to complete as expected.

```
#include <stdlib.h>

volatile int some_condition = 1;
void demo_exit1(void)
{
    /* ... Cleanup code ... */
    return;
}
void exitabnormalhandler(void)
{
    if (some_condition)
    {
        /* Clean up */
        /* Return normally */
    }
    return;
}

int demo_install_exitabnormalhandler(void)
{

    if (atexit(demo_exit1) != 0) /* demo_exit1() continues clean up */
    {
        /* Handle error */
    }
    if (atexit(exitabnormalhandler) != 0)
    {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ENV32-C

# CERT C++: ENV33-C

Do not call system()

## Description

### Rule Definition

*Do not call system().*

### Polyspace Implementation

The rule checker checks for **Unsafe call to a system function**.

## Examples

### Unsafe call to a system function

#### Issue

**Unsafe call to a system function** occurs when you use a function that invokes an implementation-defined command processor. These functions include:

- The C standard `system()` function.
- The POSIX `popen()` function.
- The Windows `_popen()` and `_wpopen()` functions.

#### Risk

If the argument of a function that invokes a command processor is not sanitized, it can cause exploitable vulnerabilities. An attacker can execute arbitrary commands or read and modify data anywhere on the system.

#### Fix

Do not use a `system`-family function to invoke a command processor. Instead, use safer functions such as POSIX `execve()` and WinAPI `CreateProcess()`.

#### Example - system() Called

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};

void func(char *arg)
{
    char buf[SIZE512];
    int retval=sprintf(buf, "/usr/bin/any_cmd %s", arg);
    if (retval<=0 || retval>SIZE512){
        /* Handle error */
        abort();
    }
    /* Use of system() to pass any_cmd with
    unsanitized argument to command processor */

    if (system(buf) == -1) { //Noncompliant
    /* Handle error */
  }
}
```

In this example, `system()` passes its argument to the host environment for the command processor to execute. This code is vulnerable to an attack by command-injection.

**Correction — Sanitize Argument and Use execve()**

In the following code, the argument of `any_cmd` is sanitized, and then passed to `execve()` for execution. `exec`-family functions are not vulnerable to command-injection attacks.

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};


void func(char *arg)
{
  char *const args[SIZE3] = {"any_cmd", arg, NULL};
  char  *const env[] = {NULL};

  /* Sanitize argument */

  /* Use execve() to execute any_cmd. */

  if (execve("/usr/bin/time", args, env) == -1) {
    /* Handle error */
  }
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

ENV33-C

# CERT C++: ENV34-C

Do not store pointers returned by certain functions

## Description

### Rule Definition

*Do not store pointers returned by certain functions.*

### Polyspace Implementation

The rule checker checks for **Misuse of return value from nonreentrant standard function**.

## Examples

### Misuse of return value from nonreentrant standard function

#### Issue

**Misuse of return value from nonreentrant standard function** occurs when these events happen in this sequence:

1. You point to the buffer returned from a nonreentrant standard function such as `getenv` or `setlocale`.

   ```
   user = getenv("USER");
   ```
2. You call that nonreentrant standard function again.

   ```
   user2 = getenv("USER2");
   ```
3. You use or dereference the pointer from the first step expecting the buffer to remain unmodified since that step. In the meantime, the call in the second step has modified the buffer.

   For instance:

   ```
   var=*user;
   ```

In some cases, the defect might appear even if you do not call the `getenv` function a second time but simply return the pointer. For instance:

```
char* func() {
     user=getenv("USER");
     .
     .
     return user;
}
```

For information on which functions are covered by this defect, see documentation on nonreentrant standard functions.

#### Risk

The C Standard allows nonreentrant functions such as `getenv` to return a pointer to a *static* buffer. Because the buffer is static, a second call to `getenv` modifies the buffer. If you continue to use the

pointer returned from the first call past the second call, you can see unexpected results. The buffer that it points to no longer has values from the first call.

The defect appears even if you do not call `getenv` a second time but simply return the pointer. The reason is that someone calling your function might use the returned pointer *after* a second call to `getenv`. By returning the pointer from your call to `getenv`, you make your function unsafe to use.

The same rationale is true for other nonreentrant functions covered by this defect.

#### Fix

After the first call to `getenv`, make a copy of the buffer that the returned pointer points to. After the second call to `getenv`, use this copy. Even if the second call modifies the buffer, your copy is untouched.

#### Example - Return from getenv Used After Second Call to getenv

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");   /* First call */
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');

        if (user_name_from_home != NULL) {
            user = getenv("USER");   /* Second call */
            if ((user != NULL) &&
                (strcmp(user, user_name_from_home) == 0)) //Noncompliant
            {
                result = 1;
            }
        }
    }
    return result;
}
```

In this example, the pointer `user_name_from_home` is derived from the pointer `home`. `home` points to the buffer returned from the first call to `getenv`. Therefore, `user_name_from_home` points to a location in the same buffer.

After the second call to `getenv`, the buffer is modified. If you continue to use `user_name_from_home`, you can get unexpected results.

#### Correction — Make Copy of Buffer Before Second Call

If you want to access the buffer from the first call to `getenv` past the second call, make a copy of the buffer after the first call. One possible correction is to use the `strdup` function to make the copy.

```
#include <stdlib.h>
#include <string.h>

int func()
{
    int result = 0;

    char *home = getenv("HOME");
    if (home != NULL) {
        char *user = NULL;
        char *user_name_from_home = strrchr(home, '/');
        if (user_name_from_home != NULL) {
            /* Make copy before second call */
            char *saved_user_name_from_home = strdup(user_name_from_home);
            if (saved_user_name_from_home != NULL) {
                user = getenv("USER");
                if ((user != NULL) &&
                    (strcmp(user, saved_user_name_from_home) == 0))
                {
                    result = 1;
                }
                free(saved_user_name_from_home);
            }
        }
    }
    return result;
}
```

### Check Information

**Group:** 49. Miscellaneous (MSC)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

ENV34-C

# CERT C++: FLP30-C

Do not use floating-point variables as loop counters

## Description

### Rule Definition

*Do not use floating-point variables as loop counters.*

### Polyspace Implementation

The rule checker checks for **Use of float variable as loop counter**.

## Examples

### Use of float variable as loop counter

#### Issue

The issue occurs when a loop counter has a floating type.

If the `for` index is a variable symbol, Polyspace checks that it is not a float.

#### Risk

When using a floating-point loop counter, accumulation of rounding errors can result in a mismatch between the expected and actual number of iterations. This rounding error can happen when a loop step that is not a power of the floating point radix is rounded to a value that can be represented by a float.

Even if a loop with a floating-point loop counter appears to behave correctly on one implementation, it can give a different number of iteration on another implementation.

#### Example - `for` Loop Counters

In this example, the three `for` loops show three different loop counters. The first and second `for` loops use float variables as loop counters, and therefore are not compliant. The third loop uses the integer `count` as the loop counter. Even though `count` is used as a float inside the loop, the variable remains an integer when acting as the loop index. Therefore, this `for` loop is compliant.

```
int main(void){
    unsigned int counter = 0u;
    int result = 0;
    float foo;

    // Float loop counters
    for(float foo = 0.0f; foo < 1.0f; foo +=0.001f){ /*Non-compliant*/
        /*counter = 1000 at the end of the loop */
        ++counter;
    }

    float fff = 0.0f;
    for(fff = 0.0f; fff <12.0f; fff += 1.0f){    /* Non-compliant*/
        result++;
    }

    // Integer loop count
    for(unsigned int count = 0u; count < 1000u; ++count){ /* Compliant */
        foo = (float) count * 0.001f;
    }
}
```

### `while` Loop Counters

This example shows two `while` loops both of which use floating point variables in the `while`-loop conditions:

- The first `while` loop uses the floating point variable `foo` in the condition and inside the loop. Because `foo` changes, floating-point rounding errors can cause unexpected behavior. Polyspace reports a violation.
- In the second while loop, the floating point array `buffer` is used in the loop condition. Polyspace identifies `iter1` and `iter2` as the loop variable. Because the loop variables are not floating point variables, a violation is not reported.

```
int main(void){
    unsigned int iter1 =0;
    int iter2;
    float foo;
    double buffer[2];
    double tmp;

    foo = 0.0f;
    while (foo < 1.0f){/* Non-compliant - foo used as a loop counter */
        foo += 0.001f;
    }

    //...
    while((iter1+1 < 2)&& (buffer[iter1]<buffer[iter2])){ //Compliant - loop counter is integer
        // swap buffer[iter1] and buffer[iter2]
        tmp = buffer[iter2];
        buffer[iter2] = buffer[iter1];
        buffer[iter1] = tmp;
        iter2 = iter1;
        iter1++;
    }

    return 1;
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**
FLP30-C

# CERT C++: FLP32-C

Prevent or detect domain and range errors in math functions

## Description

### Rule Definition

*Prevent or detect domain and range errors in math functions.*

### Polyspace Implementation

The rule checker checks for **Invalid use of standard library floating point routine**.

### Extend Checker

Extend this checker to check for defects caused by specific values and invalid use of functions from a custom library. For instance:

- You might be using a custom library of mathematical floating point functions. If a custom library function have the same domain and range as another function from the standard library, you can extend this checker to check the custom library function. See "Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries".
- A default Bug Finder analysis might not raise a defect when the input values are unknown and only a subset of inputs can cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

## Examples

### Invalid use of standard library floating point routine

#### Issue

**Invalid use of standard library floating point routine** occurs when you use invalid arguments with a floating point function from the standard library. This defect picks up:

- Rounding and absolute value routines

  `ceil, fabs, floor, fmod`
- Fractions and division routines

  `fmod, modf`
- Exponents and log routines

  `frexp, ldexp, sqrt, pow, exp, log, log10`
- Trigonometry function routines

  `cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh, atanh`

### Risk

Domain errors on standard library floating point functions result in implementation-defined values. If you use the function return value in subsequent computations, you can see unexpected results.

### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the function argument acquires invalid values. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

It is a good practice to handle for domain errors before using a standard library floating point function. For instance, before calling the `acos` function, check if the argument is in [-1.0, 1.0] and handle the error.

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and `NaNs`. Operations that results in infinities and `NaNs` might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats)`.

### Example - Arc Cosine Operation

```
#include <math.h>

double arccosine(void) {
    double degree = 5.0;
    return acos(degree); //Noncompliant
}
```

The input value to `acos` must be in the interval `[-1,1]`. This input argument, `degree`, is outside this range.

### Correction — Change Input Argument

One possible correction is to change the input value to fit the specified range. In this example, change the input value from degrees to radians to fix this defect.

```
#include <math.h>

double arccosine(void) {
    double degree = 5.0;
    double radian = degree * 3.14159 / 180.;
    return acos(radian);
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FLP32-C

# CERT C++: FLP34-C

Ensure that floating-point conversions are within range of the new type

## Description

### Rule Definition

*Ensure that floating-point conversions are within range of the new type.*

### Polyspace Implementation

This checker checks for:

- **Float conversion overflow**
- **Floating point to integer conversion overflow**

### Extend Checker

When the input values are unknown and only a subset of inputs cause an issue, Bug Finder might not detect a **Float conversion overflow** or **Floating point to integer conversion overflow**. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## Examples

### Float conversion overflow

#### Issue

**Float conversion overflow** occurs when converting a floating point number to a smaller floating point data type. If the variable does not have enough memory to represent the original number, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing conversion in subsequent computations and do not account for the overflow, you can see unexpected results.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being converted acquires its current value You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller floating point types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and `NaNs`. Operations that results in infinities and `NaNs` might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats)`.

**Example - Converting from `double` to `float`**

```
float convert(void) {

    double diam = 1e100;
    return (float)diam; //Noncompliant
}
```

In the return statement, the variable `diam` of type double (64 bits) is converted to a variable of type float (32 bits). However, the value 1^100 requires more than 32 bits to be precisely represented.

### Floating point to integer conversion overflow

**Issue**

**Floating point to integer conversion overflow** occurs when converting a floating-point value to an integer data type. If the integer part of the value cannot be represented within the storage available for the integer data type, the conversion overflows.

**Risk**

When converting from floating point to integer types, if the floating point value is outside the range that can be represented by the integer type, the behavior is undefined.

**Fix**

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

  A check for overflowing values on a `float` variable `var` can be like this:

```
if  isnan(var)
    || popcount(INT_MAX) < log2f(fabsf(var))
    || (var != 0.0F && fabsf(var) < FLT_MIN)){
    // Handle error
```

```
}
else {
    // Perform operations on var
}
```

The check determines if the floating point value is representable within an integer type:

- The value is not NaN.
- The number of bits required to store the value is less than the number of bits in `INT_MAX` (the largest integer that the `int` type can represent). The `popcount` function (not defined here) counts the number of 1's (or set bits) in a number.
- The floating point value is not lower than the smallest representable value.

**Example – Floating Point Value Converted to Integer Without Handling Overflows**

```
void func(float fVar) {
  int iVar;
  iVar = fVar; //Noncompliant
}
```

In this example, the floating point value of `fVar` is not checked for overflows before converting to an integer type. Since the argument `fVar` can contain values that are not representable within the `int` data type, the analysis flags a potential overflow.

Note that `func` is not called in this example, and the overflow is only a possibility. To see issues of these types, add the analysis option `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`.

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

FLP34-C

# CERT C++: FLP36-C

Preserve precision when converting integral values to floating-point type

## Description

### Rule Definition

*Preserve precision when converting integral values to floating-point type.*

### Polyspace Implementation

The rule checker checks for **Precision loss in integer to float conversion**.

## Examples

### Precision loss in integer to float conversion

#### Issue

**Precision loss from integer to float conversion** occurs when you cast an integer value to a floating-point type that cannot represent the original integer value.

For instance, the `long int` value `1234567890L` is too large for a variable of type `float` .

#### Risk

If the floating-point type cannot represent the integer value, the behavior is undefined (see C11 standard, 6.3.1.4, paragraph 2). For instance, least significant bits of the variable value can be dropped leading to unexpected results.

#### Fix

Convert to a floating-point type that can represent the integer value.

For instance, if the `float` data type cannot represent the integer value, use the `double` data type instead.

When writing a function that converts an integer to floating point type, before the conversion, check if the integer value can be represented in the floating-point type. For instance, `DBL_MANT_DIG * log2(FLT_RADIX)` represents the number of base-2 digits in the type `double`. Before conversion to the type `double`, check if this number is greater than or equal to the precision of the integer that you are converting. To determine the precision of an integer `num`, use this code:

```
size_t precision = 0;
while (num != 0) {
   if (num % 2 == 1) {
     precision++;
   }
   num >>= 1;
}
```

Some implementations provide a builtin function to determine the precision of an integer. For instance, GCC provides the function `__builtin_popcount`.

#### Example - Conversion of Large Integer to Floating-Point Type

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  float approx = big;//Noncompliant
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

In this example, the `long int` variable `big` is converted to `float`.

#### Correction — Use a Wider Floating-Point Type

One possible correction is to convert to the `double` data type instead of `float`.

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  double approx = big;
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

FLP36-C

# CERT C++: FLP37-C

Do not use object representations to compare floating-point values

## Description

### Rule Definition

*Do not use object representations to compare floating-point values.*

### Polyspace Implementation

The rule checker checks for **Memory comparison of float-point values**.

## Examples

### Memory comparison of float-point values

#### Issue

**Memory comparison of float-point values** occurs when you compare the object representation of floating-point values or the object representation of structures containing floating-point members. When you use the functions `memcmp`, `bcmp`, or `wmemcmp` to perform the bit pattern comparison, the defect is raised.

#### Risk

The object representation of floating-point values uses specific bit patterns to encode those values. Floating-point values that are equal, for instance `-0.0` and `0.0` in the IEC 60559 standard, can have different bit patterns in their object representation. Similarly, floating-point values that are not equal can have the same bit pattern in their object representation.

#### Fix

When you compare structures containing floating-point members, compare the structure members individually.

To compare two floating-point values, use the `==` or `!=` operators. If you follow a standard that discourages the use of these operators, such as MISRA, ensure that the difference between the floating-point values is within an acceptable range.

#### Example - Using `memcmp` to Compare Structures with Floating-Point Members

```
#include <string.h>

typedef struct {
    int i;
    float f;
} myStruct;

extern void initialize_Struct(myStruct *);

int func_cmp(myStruct *s1, myStruct *s2) {
/* Comparison between structures containing
* floating-point members */
    return memcmp //Noncompliant
        ((const void *)s1, (const void *)s2, sizeof(myStruct));
}

void func(void) {
    myStruct s1, s2;
    initialize_Struct(&s1);
    initialize_Struct(&s2);
    (void)func_cmp(&s1, &s2);
}
```

In this example, `func_cmp()` calls `memcmp()` to compare the object representations of structures `s1` and `s2`. The comparison might be inaccurate because the structures contain floating-point members.

#### Correction — Compare Structure Members Individually

One possible correction is to compare the structure members individually and to ensure that the difference between the floating-point values is within an acceptable range defined by `ESP`.

```
 #include <string.h>
#include <math.h>
typedef struct {
    int i;
    float f;
} myStruct;

extern void initialize_Struct(myStruct *);

#define ESP 0.00001

int func_cmp(myStruct *s1, myStruct *s2) {

/*Structure members are compared individually */
    return ((s1->i == s2->i) &&
            (fabsf(s1->f - s2->f) <= ESP));
}

void func(void) {
    myStruct s1, s2;
    initialize_Struct(&s1);
    initialize_Struct(&s2);
    (void)func_cmp(&s1, &s2);
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**
“Check for and Review Coding Standard Violations”

**External Websites**
FLP37-C

# CERT C++: MSC30-C

Do not use the rand() function for generating pseudorandom numbers

## Description

### Rule Definition

*Do not use the rand() function for generating pseudorandom numbers.*

### Polyspace Implementation

The rule checker checks for **Vulnerable pseudo-random number generator**.

## Examples

### Vulnerable pseudo-random number generator

#### Issue

The **Vulnerable pseudo-random number generator** identifies the use of cryptographically weak pseudo-random number generator (PRNG) routine, `rand`.

#### Risk

The `rand` function has a predictable output and must not be used for security purposes. When a predictable random value controls the execution flow, your program is vulnerable to malicious attacks.

#### Fix

Use more cryptographically sound random number generators, such as `CryptGenRandom` (Windows), `OpenSSL/RAND_bytes`(Linux/UNIX).

#### Example - Random Loop Numbers

```
#include <stdio.h>
#include <stdlib.h>

volatile int rd = 1;
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;

    nloops = rand(); //Noncompliant

    for (j = 0; j < nloops; j++) {
        if (random_r(&buf, &i))
            exit(1);
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

This example uses `rand` and `random_r` to generate random numbers. If you use these functions for security purposes, these PRNGs can be the source of malicious attacks. The CERT C checker flags the use of the `rand` function.

#### Correction — Use Stronger PRNG

One possible correction is to replace the vulnerable PRNG with a stronger random number generator.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rand.h>

volatile int rd = 1;
int main(int argc, char* argv[])
{
    int j, r, nloops;
    unsigned char buf;
    unsigned int seed;
    int i = 0;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <seed> <nloops>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    seed = atoi(argv[1]);
    nloops = atoi(argv[2]);

    for (j = 0; j < nloops; j++) {
        if (RAND_bytes(&buf, i) != 1)
            exit(1);
        printf("RAND_bytes: %u\n", (unsigned)buf);
    }
    return 0;
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

**External Websites**
MSC30-C

# CERT C++: MSC32-C

Properly seed pseudorandom number generators

## Description

### Rule Definition

*Properly seed pseudorandom number generators.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Deterministic random output from constant seed**.
- **Predictable random output from predictable seed**.

## Examples

### Deterministic random output from constant seed

#### Issue

**Deterministic random output from constant seed** detects random standard functions that when given a constant seed, have deterministic output.

#### Risk

When some random functions, such as `srand`, `srandom`, and `initstate`, have constant seeds, the results produce the same output every time that your program is run. A hacker can disrupt your program if they know how your program behaves.

#### Fix

Use a different random standard function or use a nonconstant seed.

Some standard random routines are inherently cryptographically weak on page 13-97, and should not be used for security purposes.

#### Example - Random Number Generator Initialization

```
#include <stdlib.h>

void random_num(void)
{
    srand(12345U); //Noncompliant
    /* ... */
}
```

This example initializes a random number generator using `srand` with a constant seed. The random number generation is deterministic, making this function cryptographically weak.

#### Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <stdio.h>

unsigned int random_num_time(void)
{

    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

### Predictable random output from predictable seed

#### Issue

**Predictable random output from predictable seed** looks for random standard functions that use a nonconstant but predictable seed. Examples of predictable seed generators are `time`, `gettimeofday`, and `getpid`.

#### Risk

When you use predictable seed values for random number generation, your random numbers are also predictable. A hacker can disrupt your program if they know how your program behaves.

#### Fix

You can use a different function to generate less predictable seeds.

You can also use a different random number generator that does not require a seed. For example, the Windows API function `rand_s` seeds itself by default. It uses information from the entire system, for example, system time, thread ids, system counter, and memory clusters. This information is more random and a user cannot access this information.

Some standard random routines are inherently cryptographically weak on page 13-97, and should not be used for security purposes.

#### Example - Seed as an Argument

```
#include <stdlib.h>
#include <time.h>

void seed_rng(int seed)
```

```
{
    srand(seed); //Noncompliant
}

int generate_num(void)
{
    seed_rng(time(NULL) + 3);
    /* ... */
}
```

This example uses `srand` to start the random number generator with `seed` as the seed. However, `seed` is predictable because the function `time` generates it. So, an attacker can predict the random numbers generated by `srand`.

**Correction — Use Different Random Number Generator**

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int generate_num(void)
{
    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**
MSC32-C

# CERT C++: MSC33-C

Do not pass invalid data to the asctime() function

## Description

### Rule Definition

*Do not pass invalid data to the asctime() function.*

### Polyspace Implementation

The rule checker checks for **Use of obsolete standard function**.

## Examples

### Use of obsolete standard function

#### Issue

**Use of obsolete standard function** detects calls to standard function routines that are considered legacy, removed, deprecated, or obsolete by C/C++ coding standards.

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| `asctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |
| `asctime_r` | Deprecated in POSIX.1-2008 | Implementation based on unsafe function `sprintf`. | `strftime` or `asctime_s` |
| `bcmp` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcmp` |
| `bcopy` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcpy` or `memmove` |
| `brk` and `sbrk` | Marked as legacy in SUSv2 and POSIX.1-2001. | | `malloc` |
| `bsd_signal` | Removed in POSIX.1-2008 | | `sigaction` |
| `bzero` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | `memset` |
| `ctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| ctime_r | Deprecated in POSIX.1-2008 | Implementation based on unsafe function sprintf. | strftime or asctime_s |
| cuserid | Removed in POSIX.1-2001. | Not reentrant. Precise functionality not standardized causing portability issues. | getpwuid |
| ecvt and fcvt | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | Not reentrant | snprintf |
| ecvt_r and fcvt_r | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | | snprintf |
| ftime | Removed in POSIX.1-2008 | | time, gettimeofday, clock_gettime |
| gamma, gammaf, gammal | Function not specified in any standard because of historical variations | Portability issues. | tgamma, lgamma |
| gcvt | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | snprintf |
| getcontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| getdtablesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_OPEN_MAX ) |
| gethostbyaddr | Removed in POSIX.1-2008 | Not reentrant | getaddrinfo |
| gethostbyname | Removed in POSIX.1-2008 | Not reentrant | getnameinfo |
| getpagesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_PAGESIZE ) |
| getpass | Removed in POSIX.1-2001. | Not reentrant. | getpwuid |
| getw | Not present in POSIX.1-2001. | | fread |
| getwd | Marked legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | getcwd |
| index | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strchr |
| makecontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| memalign | Appears in SunOS 4.1.3. Not in 4.4 BSD or POSIX.1-2001 | | posix_memalign |
| mktemp | Removed in POSIX.1-2008. | Generated names are predictable and can cause a race condition. | mkstemp removes race risk |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| `pthread_attr_getstackaddr` and `pthread_attr_setstackaddr` | | Ambiguities in the specification of the `stackaddr` attribute cause portability issues | `pthread_attr_getstack` and `pthread_attr_setstack` |
| `putw` | Not present in POSIX.1-2001. | Portability issues. | `fwrite` |
| `qecvt` and `qfcvt` | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | `snprintf` |
| `qecvt_r` and `qfcvt_r` | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | `snprintf` |
| `rand_r` | Marked as obsolete in POSIX.1-2008 | | |
| `re_comp` | BSD API function | Portability issues | `regcomp` |
| `re_exes` | BSD API function | Portability issues | `regexec` |
| `rindex` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | `strrchr` |
| `scalb` | Removed in POSIX.1-2008 | | `scalbln`, `scalblnf`, or `scalblnl` |
| `sigblock` | 4.3BSD signal API whose origin is unclear | | `sigprocmask` |
| `sigmask` | 4.3BSD signal API whose origin is unclear | | `sigprocmask` |
| `sigsetmask` | 4.3BSD signal API whose origin is unclear | | `sigprocmask` |
| `sigstack` | Interface is obsolete and not implemented on most platforms. | Portability issues. | `sigaltstack` |
| `sigvec` | 4.3BSD signal API whose origin is unclear | | `sigaction` |
| `swapcontext` | Removed in POSIX.1-2008 | Portability issues. | Use POSIX threads. |
| `tmpnam` and `tmpnam_r` | Marked as obsolete in POSIX.1-2008. | This function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined. | `mkstemp`, `tmpfile` |
| `ttyslot` | Removed in POSIX.1-2001. | | |
| `ualarm` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | Errors are under-specified | `setitimer` or POSIX `timer_create` |
| `usleep` | Removed in POSIX.1-2008. | | `nanosleep` |
| `utime` | SVr4, POSIX.1-2001. POSIX.1-2008 marks as obsolete. | | |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| valloc | Marked as obsolete in 4.3BSD.<br>Marked as legacy in SUSv2.<br>Removed from POSIX.1-2001 | | posix_memalign |
| vfork | Removed from POSIX.1-2008 | Under-specified in previous standards. | fork |
| wcswcs | This function was not included in the final ISO/IEC 9899:1990/ Amendment 1:1995 (E). | | wcsstr |
| WinExec | WinAPI provides this function only for 16-bit Windows compatibility. | | CreateProcess |
| LoadModule | WinAPI provides this function only for 16-bit Windows compatibility. | | CreateProcess |

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example - Printing Out Time

```
#include <stdio.h>
#include <time.h>

void timecheck_bad(int argc, char *argv[])
{
    time_t ticks;

    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks)); //Noncompliant
}
```

In this example, the function `ctime` formats the current time and prints it out. However, `ctime` was removed after C99 because it does not work on multithreaded programs.

#### Correction — Different Time Function

One possible correction is to use `strftime` instead because this function uses a set buffer size.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

void timecheck_good(int argc, char *argv[])
{
    char outBuff[1025];
    time_t ticks;
    struct tm * timeinfo;

    memset(outBuff, 0, sizeof(outBuff));

    ticks = time(NULL);
    timeinfo = localtime(&ticks);
    strftime(outBuff,sizeof(outBuff),"%I:%M%p.",timeinfo);
    fprintf(stdout, outBuff);
}
```

### Check Information

**Group:** 49. Miscellaneous (MSC)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C++ (-cert-cpp)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

MSC33-C

# CERT C++: MSC37-C

Ensure that control never reaches the end of a non-void function

## Description

### Rule Definition

*Ensure that control never reaches the end of a non-void function.*

### Polyspace Implementation

The rule checker checks for **Missing return statement**.

## Examples

### Missing return statement

#### Issue

**Missing return statement** occurs when a function does not return a value along at least one execution path. If the return type of the function is `void`, this error does not occur.

#### Risk

If a function has a non-`void` return value in its signature, it is expected to return a value. The return value of this function can be used in later computations. If the execution of the function body goes through a path where a `return` statement is missing, the function return value is indeterminate. Computations with this return value can lead to unpredictable results.

#### Fix

In most cases, you can fix this defect by placing the `return` statement at the end of the function body.

Alternatively, you can identify which execution paths through the function body do not have a `return` statement and add a `return` statement on those paths. Often the result details show a sequence of events that indicate this execution path. You can add a `return` statement at an appropriate point in the path. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface".

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example - Missing or invalid return statement error**

```
int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }
 }  //Noncompliant
/* Defect: No return value if n is not 0*/
```

If `n` is equal to 0, the code does not enter the `if` statement. Therefore, the function `AddSquares` does not return a value if `n` is 0.

**Correction — Place Return Statement on Every Execution Path**

One possible correction is to return a value in every branch of the `if...else` statement.

```
 int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }

   /*Fix: Place a return statement on branches of if-else */
   else
     return 0;
  }
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**

"Check for and Review Coding Standard Violations"

### External Websites

MSC37-C

# CERT C++: MSC38-C

Do not treat a predefined identifier as an object if it might only be implemented as a macro

## Description

### Rule Definition

*Do not treat a predefined identifier as an object if it might only be implemented as a macro.*

### Polyspace Implementation

The rule checker checks for **Predefined macro used as an object**.

## Examples

### Predefined macro used as an object

#### Issue

**Predefined macro used as an object** occurs when you use certain identifiers in a way that requires an underlying object to be present. These identifiers are defined as macros. The C Standard does not allow you to redefine them as objects. You use the identifiers in such a way that macro expansion of the identifiers cannot occur.

For instance, you refer to an external variable `errno`:

```
extern int errno;
```

However, `errno` does not occur as a variable but a macro.

The defect applies to these macros: `assert`, `errno`, `math_errhandling`, `setjmp`, `va_arg`, `va_copy`, `va_end`, and `va_start`. The checker looks for the defect only in source files (not header files).

#### Risk

The C11 Standard (Sec. 7.1.4) allows you to redefine most macros as objects. To access the object and not the macro in a source file, you do one of these:

- Redeclare the identifier as an external variable or function.
- For function-like macros, enclose the identifier name in parentheses.

If you try to use these strategies for macros that cannot be redefined as objects, an error occurs.

#### Fix

Do not use the identifiers in such a way that a macro expansion is suppressed.

- Do not redeclare the identifiers as external variables or functions.
- For function-like macros, do not enclose the macro name in parentheses.

#### Example - Use of `assert` as Function

```
#include<assert.h>
typedef void (*err_handler_func)(int);

extern void demo_handle_err(err_handler_func, int);

void func(int err_code) {
    extern void assert(int);    //Noncompliant
    demo_handle_err(&(assert), err_code); //Noncompliant
}
```

In this example, the `assert` macro is redefined as an external function. When passed as an argument to `demo_handle_err`, the identifier `assert` is enclosed in parentheses, which suppresses use of the `assert` macro.

#### Correction — Use `assert` as Macro

One possible correction is to directly use the `assert` macro from `assert.h`. A different implementation of the function `demo_handle_err` directly uses the `assert` macro instead of taking the address of an `assert` function.

```
#include<assert.h>
void demo_handle_err(int err_code) {
    assert(err_code == 0);
}

void func(int err_code) {
    demo_handle_err(err_code);
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

MSC38-C

# CERT C++: MSC39-C

Do not call va_arg() on a va_list that has an indeterminate value

## Description

### Rule Definition

*Do not call va_arg() on a va_list that has an indeterminate value.*

### Polyspace Implementation

The rule checker checks for **Use of indeterminate va_list values**.

## Examples

### Use of indeterminate va_list values

#### Issue

This issue occurs when:

- You use a local `va_list` without initializing it first using `va_start` or `va_copy`.

  You might be using the local `va_list` in `va_arg` or a `vprintf`-like function (function that takes variable number of arguments).
- You use a `va_list` (variable argument list) from a function parameter directly instead of making a copy using `va_copy` and using the copy.

Note that the checker works on a per-function basis. If you initialize a `va_list` with `va_start` within a block, the checker considers the list as initialized beyond the block for the remainder of the function. Likewise, if you end the list with `va_end` within a block, the checker considers the list as ended beyond the block for the remainder of the function.

#### Risk

If you use a local `va_list` without initializing it first, the behavior is undefined.

If you pass a `va_list` to another function and use it there, the `va_list` has indeterminate values in the original calling function. Using the `va_list` in the calling function following the function call can produce unexpected results.

#### Fix

Initialize a local `va_list` with `va_start` or `va_copy` before using it.

Pass a `va_list` by reference. In the called function, make a copy of the passed `va_list` and use the copy. You can then continue to access the original `va_list` in the calling function.

#### Example – Direct Use of `va_list` From Another Function

```
#include <cstdarg>
#include <cstdio>
#include <climits>
```

```
int containsOutliers(size_t count, va_list ap) {
  for (size_t i = 1; i < count; ++i) {
    if (va_arg(ap, int) > INT_MAX) { //Noncompliant
      return 1;
    }
  }
  return 0;
}

int printList(size_t count, ...) {
  va_list ap;
  va_start(ap, count);

  if (containsOutliers(count, ap)) {
    va_end(ap);
    return 1;
  }

  for (size_t i = 0; i < count; ++i) {
    printf("%d", va_arg(ap, int));
  }

  va_end(ap);
  return 0;
}
```

In this example, the checker flags the direct use of the `va_list` variable `ap` obtained as argument in the `containsOutliers` function.

**Correction – Copy `va_list` Obtained from Another Function**

To avoid the violation, pass the `va_list` by reference and make a copy of the variable in the `containsOutliers` function. Perform further operations on the copy.

```
#include <cstdarg>
#include <cstdio>
#include <climits>

int containsOutliers(size_t count, va_list* ap) {
  va_list copiedAp;
  va_copy (copiedAp, *ap);

  for (size_t i = 1; i < count; ++i) {
    if (va_arg(copiedAp, int) > INT_MAX) {
      return 1;
    }
  }
  return 0;
}

int printList(size_t count, ...) {
  va_list ap;
  va_start(ap, count);

  if (containsOutliers(count, &ap)) {
    va_end(ap);
    return 1;
  }

  for (size_t i = 0; i < count; ++i) {
    printf("%d", va_arg(ap, int));
  }

  va_end(ap);
  return 0;
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC39-C

# CERT C++: MSC40-C

Do not violate constraints

## Description

### Rule Definition

*Do not violate constraints.*

### Polyspace Implementation

The rule checker checks for **Inline constraint not respected**.

## Examples

### Inline constraint not respected

#### Issue

**Inline constraint not respected** occurs when you refer to a file scope modifiable static variable or define a local modifiable static variable in a nonstatic inlined function. The checker considers a variable as modifiable if it is not `const`-qualified.

For instance, `var` is a modifiable `static` variable defined in an `inline` function `func`. `g_step` is a file scope modifiable static variable referred to in the same inlined function.

```
static int g_step;
inline void func (void) {
   static int var = 0;
   var += g_step;
}
```

#### Risk

When you modify a static variable in multiple function calls, you expect to modify the same variable in each call. For instance, each time you call `func`, the same instance of `var1` is incremented but a separate instance of `var2` is incremented.

```
void func(void) {
   static var1 = 0;
   int var2 = 0;
   var1++;
   var2++;
}
```

If a function has an inlined and non-inlined definition (in separate files), when you call the function, the C standard allows compilers to use either the inlined or the non-inlined form (see ISO/IEC 9899:2011, sec. 6.7.4). If your compiler uses an inlined definition in one call and the non-inlined definition in another, you are no longer modifying the same variable in both calls. This behavior defies the expectations from a static variable.

#### Fix

Use one of these fixes:

- If you do not intend to modify the variable, declare it as `const`.

  If you do not modify the variable, there is no question of unexpected modification.
- Make the variable non-`static`. Remove the `static` qualifier from the declaration.

  If the variable is defined in the function, it becomes a regular local variable. If defined at file scope, it becomes an extern variable. Make sure that this change in behavior is what you intend.
- Make the function `static`. Add a `static` qualifier to the function definition.

  If you make the function `static`, the file with the inlined definition always uses the inlined definition when the function is called. Other files use another definition of the function. The question of which function definition gets used is not left to the compiler.

#### Example - Static Variable Use in Inlined and External Definition

```
/* file1. c  : contains inline definition of get_random()*/

inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;  //Noncompliant
    static unsigned int m_w = 0xbaddecaf;  //Noncompliant

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}
```

```
/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

In this example, `get_random()` has an inline definition in `file1.c` and an external definition in `file2.c`. When `get_random` is called in `file1.c`, compilers are free to choose whether to use the inline or the external definition.

Depending on the definition used, you might or might not modify the version of `m_z` and `m_w` in the inlined version of `get_random()`. This behavior contradicts the usual expectations from a static variable. When you call `get_random()`, you expect to always modify the same `m_z` and `m_w`.

**Correction — Make Inlined Function Static**

One possible correction is to make the inlined `get_random()` static. Irrespective of your compiler, calls to `get_random()` in `file1.c` then use the inlined definition. Calls to `get_random()` in other files use the external definition. This fix removes the ambiguity about which definition is used and whether the static variables in that definition are modified.

```
/* file1. c  : contains inline definition of get_random()*/

static inline unsigned int get_random(void)
{

    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}


int call_get_random(void)
{
    unsigned int rand_no;
    int ii;
    for (ii = 0; ii < 100; ii++) {
         rand_no = get_random();
    }
    rand_no = get_random();
    return 0;
}

/* file2. c  : contains external definition of get_random()*/

extern unsigned int get_random(void)
{
    /* Initialize seeds */
    static unsigned int m_z = 0xdeadbeef;
    static unsigned int m_w = 0xbaddecaf;

    /* Compute next pseudorandom value and update seeds */
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MSC40-C

# CERT C++: MSC41-C

Never hard code sensitive information

## Description

### Rule Definition

*Never hard code sensitive information.*

### Polyspace Implementation

The rule checker checks for **Hard coded sensitive data**.

## Examples

### Hard coded sensitive data

**Hard coded sensitive data** occurs when data that is potentially sensitive is directly exposed in the code, for instance, as string literals. The checker identifies data as sensitive from their use in certain functions such as password encryption functions.

Following data can be potentially sensitive.

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Host name | • `sethostname`, `setdomainname`, `gethostbyname`, `gethostbyname2`, `getaddrinfo`, `gethostbyname_r`, `gethostbyname2_r` (string argument)<br>• `inet_aton`, `inet_pton`, `inet_net_pton`, `inet_addr`, `inet_network` (string argument)<br>• `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (2nd argument) |
| Password | • `CreateProcessWithLogonW`, `LogonUser` (1st argument)<br>• `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (3rd argument) |

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Database | • MySQL: `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (4th argument)<br>• SQLite: `sqlite3_open`, `sqlite3_open16`, `sqlite3_open_v2` (1st argument)<br>• PostgreSQL: `PQconnectdb`<br>• Microsoft SQL: `SQLDriverConnect` (3rd argument) |
| User name | • `getpw`, `getpwnam`, `getpwnam_r`, `getpwuid`, `getpwuid_r` |
| Salt | `crypt`, `crypt_r` (2nd argument) |
| Cryptography keys and initialization vectors | OpenSSL:<br>• `EVP_CipherInit`, `EVP_EncryptInit`, `EVP_DecryptInit` (3rd argument)<br>• `EVP_CipherInit_ex`, `EVP_EncryptInit_ex`, `EVP_DecryptInit_ex` (4th argument) |
| Seed | • `srand`, `srandom`, `initstate` (1st argument)<br>• OpenSSL: `RAND_seed`, `RAND_add` |

#### Risk

Information that is hardcoded can be queried from binaries generated from the code.

#### Fix

Avoid hard coding sensitive information.

### Check Information

**Group:** Rule 48. Miscellaneous (MSC)

## Version History

**Introduced in R2020a**

### See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

MSC41-C

# CERT C++: MSC50-CPP

Do not use std::rand() for generating pseudorandom numbers

## Description

### Rule Definition

*Do not use std::rand() for generating pseudorandom numbers.*

### Polyspace Implementation

The rule checker checks for **Vulnerable pseudo-random number generator**.

## Examples

### Vulnerable pseudo-random number generator

#### Issue

The **Vulnerable pseudo-random number generator** identifies uses of cryptographically weak pseudo-random number generator (PRNG) routines.

The list of cryptographically weak routines flagged by this checker include:

- `rand`, `random`
- `drand48`, `lrand48`, `mrand48`, `erand48`, `nrand48`, `jrand48`, and their `_r` equivalents such as `drand48_r`
- `RAND_pseudo_bytes`

#### Risk

These cryptographically weak routines are predictable and must not be used for security purposes. When a predictable random value controls the execution flow, your program is vulnerable to malicious attacks.

#### Fix

Use more cryptographically sound random number generators, such as `CryptGenRandom` (Windows), `OpenSSL/RAND_bytes`(Linux/UNIX).

#### Example - Random Loop Numbers

```
#include <stdio.h>
#include <stdlib.h>

volatile int rd = 1;
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;
    nloops = rand(); //Noncompliant

    for (j = 0; j < nloops; j++) {
        if (random_r(&buf, &i)) //Noncompliant
            exit(1);
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

This example uses `rand` and `random_r` to generate random numbers. If you use these functions for security purposes, these PRNGs can be the source of malicious attacks.

**Correction — Use Stronger PRNG**

One possible correction is to replace the vulnerable PRNG with a stronger random number generator.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rand.h>

volatile int rd = 1;
int main(int argc, char* argv[])
{
    int j, r, nloops;
    unsigned char buf;
    unsigned int seed;
    int i = 0;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <seed> <nloops>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    seed = atoi(argv[1]);
    nloops = atoi(argv[2]);

    for (j = 0; j < nloops; j++) {
        if (RAND_bytes(&buf, i) != 1)
            exit(1);
        printf("RAND_bytes: %u\n", (unsigned)buf);
    }
    return 0;
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

MSC50-CPP

# CERT C++: MSC51-CPP

Ensure your random number generator is properly seeded

## Description

### Rule Definition

*Ensure your random number generator is properly seeded.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Deterministic random output from constant seed**.
- **Predictable random output from predictable seed**.

## Examples

### Deterministic random output from constant seed

#### Issue

**Deterministic random output from constant seed** detects random standard functions that when given a constant seed, have deterministic output.

#### Risk

When some random functions, such as `srand`, `srandom`, and `initstate`, have constant seeds, the results produce the same output every time that your program is run. A hacker can disrupt your program if they know how your program behaves.

#### Fix

Use a different random standard function or use a nonconstant seed.

Some standard random routines are inherently cryptographically weak on page 13-97, and should not be used for security purposes.

#### Example - Random Number Generator Initialization

```
#include <stdlib.h>

void random_num(void)
{
    srand(12345U); //Noncompliant
    /* ... */
}
```

This example initializes a random number generator using `srand` with a constant seed. The random number generation is deterministic, making this function cryptographically weak.

#### Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <stdio.h>

unsigned int random_num_time(void)
{

    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

### Predictable random output from predictable seed

#### Issue

**Predictable random output from predictable seed** looks for random standard functions that use a nonconstant but predictable seed. Examples of predictable seed generators are `time`, `gettimeofday`, and `getpid`.

#### Risk

When you use predictable seed values for random number generation, your random numbers are also predictable. A hacker can disrupt your program if they know how your program behaves.

#### Fix

You can use a different function to generate less predictable seeds.

You can also use a different random number generator that does not require a seed. For example, the Windows API function `rand_s` seeds itself by default. It uses information from the entire system, for example, system time, thread ids, system counter, and memory clusters. This information is more random and a user cannot access this information.

Some standard random routines are inherently cryptographically weak on page 13-97, and should not be used for security purposes.

#### Example - Seed as an Argument

```
#include <stdlib.h>
#include <time.h>

void seed_rng(int seed)
```

```
{
    srand(seed); //Noncompliant
}

int generate_num(void)
{
    seed_rng(time(NULL) + 3);
    /* ... */
}
```

This example uses `srand` to start the random number generator with `seed` as the seed. However, `seed` is predictable because the function `time` generates it. So, an attacker can predict the random numbers generated by `srand`.

**Correction — Use Different Random Number Generator**

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int generate_num(void)
{
    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**
MSC51-CPP

# CERT C++: MSC52-CPP

Value-returning functions must return a value from all exit paths

## Description

### Rule Definition

*Value-returning functions must return a value from all exit paths.*

### Polyspace Implementation

The rule checker checks for **Missing return statement**.

## Examples

### Missing return statement

#### Issue

**Missing return statement** occurs when a function does not return a value along at least one execution path. If the return type of the function is `void`, this error does not occur.

#### Risk

If a function has a non-`void` return value in its signature, it is expected to return a value. The return value of this function can be used in later computations. If the execution of the function body goes through a path where a `return` statement is missing, the function return value is indeterminate. Computations with this return value can lead to unpredictable results.

#### Fix

In most cases, you can fix this defect by placing the `return` statement at the end of the function body.

Alternatively, you can identify which execution paths through the function body do not have a `return` statement and add a `return` statement on those paths. Often the result details show a sequence of events that indicate this execution path. You can add a `return` statement at an appropriate point in the path. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface".

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example - Missing or invalid return statement error**

```
int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }
 }  //Noncompliant
/* Defect: No return value if n is not 0*/
```

If `n` is equal to 0, the code does not enter the `if` statement. Therefore, the function `AddSquares` does not return a value if `n` is 0.

**Correction — Place Return Statement on Every Execution Path**

One possible correction is to return a value in every branch of the `if...else` statement.

```
 int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }

   /*Fix: Place a return statement on branches of if-else */
   else
     return 0;
  }
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**
MSC52-CPP

# CERT C++: MSC53-CPP

Do not return from a function declared [[noreturn]]

## Description

### Rule Definition

*Do not return from a function declared [[noreturn]]*

### Polyspace Implementation

The rule checker checks for **`[[noreturn]]` functions returning to Caller**.

## Examples

### `[[noreturn]]` functions returning to caller

#### Issue

This defect occurs when a `[[noreturn]]` function eventually returns the flow of execution to the caller function. The compiler expects that a function declared by using the `[[noreturn]]` attribute does not return the flow of execution. That is, if a `[[noreturn]]` function `f()` is called from `main()`, then the compiler expects that the flow of execution is not returned to `main()`. If such a function eventually returns the flow of execution, it leads to undefined behavior.

#### Risk

If a `[[noreturn]]` function eventually returns the flow of execution, it leads to undefined behavior, which can be exploited to cause data integrity violations.

#### Fix

If a function has no `return` statement, then the final closing brace of a function implies an implicit `return`. Omitting a `return` statement in the function does not prevent the flow of execution from returning. A `[[noreturn]]` function can prohibit returning the flow of execution to the calling function by:

- Entering an infinite loop
- Raising an exception
- Calling another `[[noreturn]]` function

#### Example

Consider the following code containing the function `noncompliant()`, which is declared as `[[noreturn]]`.

```
#include <cstdlib>
[[noreturn]] void bad_f(int i)
{
    if (i > 0)
    throw "Received positive input";
    else if (i < 0)
    std::exit(0);
} //Noncompliant
```

When the input `i` is zero, the flow of execution skips the `if-else-if` block of code and returns to the caller implicitly. Because the `[[noreturn]]` function returns the flow of execution in a code path, this function is noncompliant with this rule.

**Correction**

A `[[noreturn]]` function must not return the flow of execution in a code path. You can prevent returning in several ways. Consider the following code where the `[[noreturn]]` function does not return the flow of execution in a code path.

```
#include <cstdlib>
[[noreturn]] void compliant(int i)
{
    if (i > 0)
    throw "Received positive input";
    else if (i < 0)
    std::exit(0);
    else if(i==0)
    while(true){
        //...
    }
}//Compliant
```

This function is compliant with this rule because:

- When `i > 0`, the function raises an exception.
- When `i < 0`, the function calls the `[[noreturn]]` function `std::exit()`.
- When `i==0`, the function enters an infinite loop.

Because the `[[noreturn]]` function does not return the flow of execution in a call path, it is compliant with this rule.

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2020b**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

MSC53-CPP

# CERT C++: PRE30-C

Do not create a universal character name through concatenation

## Description

### Rule Definition

*Do not create a universal character name through concatenation.*

### Polyspace Implementation

The rule checker checks for **Universal character name from token concatenation**.

## Examples

### Universal character name from token concatenation

#### Issue

**Universal character name from token concatenation** occurs when two preprocessing tokens joined with a ## operator create a universal character name. A universal character name begins with `\u` or `\U` followed by hexadecimal digits. It represents a character not found in the basic character set.

For instance, you form the character `\u0401` by joining two tokens:

```
#define assign(uc1, uc2, val) uc1##uc2 = val
...
assign(\u04, 01, 4);
```

#### Risk

The C11 Standard (Sec. 5.1.1.2) states that if a universal character name is formed by token concatenation, the behavior is undefined.

#### Fix

Use the universal character name directly instead of producing it through token concatenation.

#### Example - Universal Character Name from Token Concatenation

```
#define assign(uc1, uc2, val) uc1##uc2 = val

int func(void) {
    int \u0401 = 0;
    assign(\u04, 01, 4);
    return \u0401;
}
```

In this example, the `assign` macro, when expanded, joins the two tokens `\u04` and `01` to form the universal character name `\u0401`.

#### Correction — Use Universal Character Name Directly

One possible correction is to use the universal character name `\u0401` directly. The correction redefines the `assign` macro so that it does not join tokens.

```
#define assign(ucn, val) ucn = val

int func(void) {
    int \u0401 = 0;
    assign(\u0401, 4);
    return \u0401;
}
```

### Check Information

**Group:** 49. Miscellaneous (MSC)

## Version History

**Introduced in R2019a**

### See Also

`Check SEI CERT-C++ (-cert-cpp))`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

PRE30-C

# CERT C++: PRE31-C

Avoid side effects in arguments to unsafe macros

## Description

### Rule Definition

*Avoid side effects in arguments to unsafe macros.*

### Polyspace Implementation

The rule checker checks for **Side effect in arguments to unsafe macro**.

## Examples

### Side effect in arguments to unsafe macro

#### Issue

**Side effect in arguments to unsafe macro** occurs when you call an unsafe macro with an expression that has a side effect.

- *Unsafe macro*: When expanded, an unsafe macro evaluates its arguments multiple times or does not evaluate its argument at all.

  For instance, the ABS macro evaluates its argument x twice.

  ```
  #define ABS(x) (((x) < 0) ? -(x) : (x))
  ```
- *Side effect*: When evaluated, an expression with a side effect modifies at least one of the variables in the expression.

  For instance, ++n modifies n, but n+1 does not modify n.

  The checker does not consider side effects in nested macros. The checker also does not consider function calls or volatile variable access as side effects.

#### Risk

If you call an unsafe macro with an expression that has a side effect, the expression is evaluated multiple times or not evaluated at all. The side effect can occur multiple times or not occur at all, causing unexpected behavior.

For instance, in the call MACRO(++n), you expect only one increment of the variable n. If MACRO is an unsafe macro, the increment happens more than once or does not happen at all.

The checker flags expressions with side effects in the assert macro because the assert macro is disabled in non-debug mode. To compile in non-debug mode, you define the NDEBUG macro during compilation. For instance, in GCC, you use the flag -DNDEBUG.

#### Fix

Evaluate the expression with a side effect in a separate statement, and then use the result as a macro argument.

For instance, instead of:

```
MACRO(++n);
```

perform the operation in two steps:

```
++n;
MACRO(n);
```

Alternatively, use an inline function instead of a macro. Pass the expression with side effect as argument to the inline function.

The checker considers modifications of a local variable defined only in the block scope of a macro body as a side effect. This defect cannot happen since the variable is visible only in the macro body. If you see a defect of this kind, ignore the defect.

#### Example - Macro Argument with Side Effects

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  int m = ABS(++n); //Noncompliant

  /* ... */
}
```

In this example, the ABS macro evaluates its argument twice. The second evaluation can result in an unintended increment.

#### Correction — Separate Evaluation of Expression from Macro Usage

One possible correction is to first perform the increment, and then pass the result to the macro.

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int n) {
  /* Validate that n is within the desired range */
  ++n;
  int m = ABS(n);

  /* ... */
}
```

#### Correction — Evaluate Expression in Inline Function

Another possible correction is to evaluate the expression in an inline function.

```
static inline int iabs(int x) {
  return (((x) < 0) ? -(x) : (x));
}

void func(int n) {
  /* Validate that n is within the desired range */

int m = iabs(++n);

  /* ... */
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

PRE31-C

# CERT C++: PRE32-C

Do not use preprocessor directives in invocations of function-like macros

## Description

### Rule Definition

*Do not use preprocessor directives in invocations of function-like macros.*

### Polyspace Implementation

The rule checker checks for **Preprocessor directive in macro argument**.

## Examples

### Preprocessor directive in macro argument

#### Issue

**Preprocessor directive in macro argument** occurs when you use a preprocessor directive in the argument to a function-like macro or a function that might be implemented as a function-like macro.

For instance, a `#ifdef` statement occurs in the argument to a `memcpy` function. The `memcpy` function might be implemented as a macro.

```
memcpy(dest, src,
    #ifdef PLATFORM1
      12
    #else
      24
    #endif
  );
```

The checker flags similar usage in `printf` and `assert`, which can also be implemented as macros.

#### Risk

During preprocessing, a function-like macro call is replaced by the macro body and the parameters are replaced by the arguments to the macro call (argument substitution). Suppose a macro `min()` is defined as follows.

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you call `min(1,2)`, it is replaced by the body `((X) < (Y) ? (X) : (Y))`. `X` and `Y` are replaced by 1 and 2.

According to the C11 Standard (Sec. 6.10.3), if the list of arguments to a function-like macro itself has preprocessing directives, the argument substitution during preprocessing is undefined.

#### Fix

To ensure that the argument substitution happens in an unambiguous manner, use the preprocessor directives outside the function-like macro.

For instance, to execute memcpy with different arguments based on a #ifdef directive, call memcpy multiple times within the #ifdef directive branches.

```
#ifdef PLATFORM1
    memcpy(dest, src, 12);
#else
    memcpy(dest, src, 24);
#endif
```

**Example - Directives in Function-Like Macros**

```
#include <stdio.h>

#define print(A) printf(#A)

void func(void) {
    print(
#ifdef SW //Noncompliant
          "Message 1"
#else
          "Message 2"
#endif //Noncompliant
         );
}
```

In this example, the preprocessor directives #ifdef and #endif occur in the argument to the function-like macro print().

**Correction — Use Directives Outside Macro**

One possible correction is to use the function-like macro multiple times in the branches of the #ifdef directive.

```
#include <stdio.h>

#define print(A) printf(#A)

void func(void) {
#ifdef SW
        print("Message 1");
#else
        print("Message 2");
#endif
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

Check SEI CERT-C++ (-cert-cpp))

**Topics**
“Check for and Review Coding Standard Violations”

**External Websites**
PRE32-C

# CERT C++: SIG31-C

Do not access shared objects in signal handlers

## Description

### Rule Definition

*Do not access shared objects in signal handlers.*

### Polyspace Implementation

The rule checker checks for **Shared data access within signal handler**.

## Examples

### Shared data access within signal handler

#### Issue

**Shared data access within signal handler** occurs when you access or modify a shared object inside a signal handler.

#### Risk

When you define a signal handler function to access or modify a shared object, the handler accesses or modifies the shared object when it receives a signal. If another function is already accessing the shared object, that function causes a race condition and can leave the data in an inconsistent state.

#### Fix

To access or modify shared objects inside a signal handler, check that the objects are lock-free atomic, or, if they are integers, declare them as `volatile sig_atomic_t`.

#### Example - `int` Variable Access in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* declare global variable. */
int e_flag;

void sig_handler(int signum)
{
    /* Signal handler accesses variable that is not
    of type volatile sig_atomic_t. */
    e_flag = signum;  //Noncompliant
}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` accesses `e_flag`, a variable of type `int`. A concurrent access by another function can leave `e_flag` in an inconsistent state.

**Correction — Declare Variable of Type `volatile sig_atomic_t`**

Before you access a shared variable from a signal handler, declare the variable with type `volatile sig_atomic_t` instead of `int`. You can safely access variables of this type asynchronously.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* Declare variable of type volatile sig_atomic_t. */
volatile sig_atomic_t e_flag;
void sig_handler(int signum)
{
    /* Use variable of proper type inside signal handler. */
    e_flag = signum;

}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

SIG31-C

# CERT C++: SIG34-C

Do not call signal() from within interruptible signal handlers

## Description

### Rule Definition

*Do not call signal() from within interruptible signal handlers.*

### Polyspace Implementation

The rule checker checks for **Signal call from within signal handler**.

## Examples

### Signal call from within signal handler

#### Issue

**Signal call from within signal handler** occurs when you call `signal()` from a signal handler on Windows platforms.

The issue is detected only if you specify a Visual Studio compiler. See `Compiler (-compiler)`.

#### Risk

The function `signal()` associates a signal with a signal handler function. On platforms such as Windows, which removes this association after receiving the signal, you might call the function `signal()` again *within the signal handler* to re-establish the association.

However, this attempt to make a signal handler persistent is prone to race conditions. On Windows platforms, from the time the signal handler begins execution to when the `signal` function is called again, it is the default signal handling, `SIG_DFL`, that is active. If a second signal is received within this time window, you see the default signal handling and not the custom signal handler, but you might expect otherwise.

#### Fix

Do not call `signal()` from a signal handler on Windows platforms.

#### Example - `signal()` Called from Signal Handler

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;

void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;

    /* Call signal() to reestablish sig_handler
    upon receiving SIG_ERR. */

    if (signal(s0, sig_handler) == SIG_ERR)  //Noncompliant
    {
        /* Handle error */
    }
}

void func(void)
{
        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
  /* more code */
}
```

In this example, the definition of `sig_handler()` includes a call to `signal()` when the handler catches `SIG_ERR`. On Windows platforms, signal handlers are nonpersistent. This code can result in a race condition.

The issue is detected only if you specify a compiler such as `visual15.x` for the analysis.

**Correction — Do Not Call `signal()` from Signal Handler**

Avoid attempting to make a signal handler persistent on Windows. If your code requires the use of a persistent signal handler on a Windows platform, use a persistent signal handler after performing a thorough risk analysis.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;


void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;
    /* No call to signal() */
}

int main(void)
{

        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

# Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

SIG34-C

# CERT C++: SIG35-C

Do not return from a computational exception signal handler

## Description

### Rule Definition

*Do not return from a computational exception signal handler.*

### Polyspace Implementation

The rule checker checks for **Return from computational exception signal handler**.

## Examples

### Return from computational exception signal handler

#### Issue

**Return from computational exception signal handler** occurs when a signal handler returns after catching a computational exception signal `SIGFPE`, `SIGILL`, or `SIGSEGV`.

#### Risk

A signal handler that returns normally from a computational exception is undefined behavior. Even if the handler attempts to fix the error that triggered the signal, the program can behave unexpectedly.

#### Fix

Check the validity of the values of your variables before the computation to avoid using a signal handler to catch exceptions. If you cannot avoid a handler to catch computation exception signals, call `abort()`, `quick_exit()`, or `_Exit()` in the handler to stop the program.

#### Example - Signal Handler Return from Division by Zero

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

static volatile sig_atomic_t denom;
/* Declare signal handler to catch division by zero
computation error. */
void sig_handler(int s)
{
    int s0 = s;
    if (denom == 0)
    {
        denom = 1;
    }
    /* Normal return from computation exception
    signal */
    return;  //Noncompliant
}
```

```
long func(int v)
{
    denom = (sig_atomic_t)v;

        if (signal(SIGFPE, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }

    long result = 100 / (long)denom;
    return result;
}
```

In this example, `sig_handler` is declared to handle a division by zero computation error. The handler changes the value of `denom` if it is zero and returns, which is undefined behavior.

**Correction — Call `abort()` to Terminate Program**

After catching a computational exception, call `abort()` from `sig_handler` to exit the program without further error.

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

static volatile sig_atomic_t denom;
/* Declare signal handler to catch division by zero
computation error. */

void sig_handler(int s)
{
    int s0 = s;
    /* call to abort() to exit the program */
    abort();
}

long func(int v)
{
    denom = (sig_atomic_t)v;

        if (signal(SIGFPE, sig_handler) == SIG_ERR)
        {
            /* Handle error */
        }

    long result = 100 / (long)denom;
    return result;
}
```

## Check Information

**Group:** 49. Miscellaneous (MSC)

## Version History

**Introduced in R2019a**

## See Also

`Check SEI CERT-C++ (-cert-cpp))`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

SIG35-C

# AUTOSAR C++14 Rules

# AUTOSAR C++14 Rule A0-1-1

A project shall not contain instances of non-volatile variables being given values that are not subsequently used

## Description

### Rule Definition

*A project shall not contain instances of non-volatile variables being given values that are not subsequently used.*

### Rationale

If you assign a value to a variable but do not use the variable value subsequently, the assignment might indicate a programming error. Perhaps you forgot to use the variable later or incorrectly used other variables at the intended points of use.

### Polyspace Implementation

The checker flags value assignments to local and static variables with file scope if the assigned values are not subsequently used. The checker considers `const`-qualified global variables without the `extern` specifier as static variables with file scope.

The checker flags:

- Initializations if the initialized variable is not used.
- Non-initialization assignments if the assigned values are not used.

The checker does not flag the situation where an initialization value is immediately overwritten and therefore ends up unused.

The checker does not flag redundant assignments:

- To variables with class type.
- In the last iteration of a loop, if the assignments in the previous iterations are not redundant.

  For instance, the assignment `prevIter = i` in the last iteration of the loop is redundant but the assignments in the previous iterations are not.

  ```
  void doSomething(int);

  void func() {
    int prevIter=-1, uBound=100;
    for(int i=0; i < uBound; i++) {
          doSomething(prevIter);
          prevIter = i;
    }
  }
  ```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Initialization Value Not Used

```
class largeInteger {
       largeInteger(int d1, int d2, int d3):
            lastFiveDigits(d1), nextFiveDigits(d2), firstFiveDigits(d3){}
       largeInteger& operator=(const largeInteger& other) {
            if(&other !=this) {
              firstFiveDigits = other.firstFiveDigits;
              nextFiveDigits = other.nextFiveDigits;
              lastFiveDigits = other.lastFiveDigits;
            }
            return *this;
       }
       void printIntegerValue();
    private:
        int firstFiveDigits;
        int nextFiveDigits;
        int lastFiveDigits;
};

bool compareValues(largeInteger, largeInteger);

void func() {
    largeInteger largeUnit{10000,0,0}; //Compliant
    largeInteger smallUnit{1,0,0}; //Compliant
    largeInteger tinyUnit{0,1,0}; //Noncompliant
    if(compareValues (largeUnit, smallUnit)) {
        //Perform some action
    }
}
```

In this example, the variable `tinyUnit` is initialized but never used.

## Check Information

**Group:** Language independent issues
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A0-1-2

The value returned by a function having a non-void return type that is not an overloaded operator shall be used

## Description

### Rule Definition

*The value returned by a function having a non-void return type that is not an overloaded operator shall be used.*

### Rationale

The unused return value can indicate a coding error or oversight.

Overloaded operators are excluded from this rule because their usage must emulate built-in operators that might not use their return value.

### Polyspace Implementation

The rule checker reports a violation on functions with a non-`void` return type if the return value is not used or not explicitly cast to a `void` type.

The checker does not flag the functions `memcpy`, `memset`, `memmove`, `strcpy`, `strncpy`, `strcat`, `strncat` because these functions simply return a pointer to their first arguments.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unused Return Value

```
#include <iostream>
#include <new>
#include <algorithm>
#include <cstdint>
#include <vector>

int assignMemory(int * ptr){
     int res = 1;
     ptr = new (std::nothrow) int;
     if(ptr==NULL) {
         res = 0;
     }
     return res;
}
void foo()
{
}

void main() {
    int val;
    int status;
    std::vector<std::int8_t> numVec{10,10,20,20,30,40,50,50,60};

    assignMemory(&val);   //Noncompliant
    status = assignMemory(&val); //Compliant
    (void)assignMemory(&val); //Compliant

    numVec.erase(std::unique(numVec.begin(), numVec.end()), numVec.end());// Noncompliant
}
```

In this example, Polyspace flags calls to functions with unused return value:

- Because `main` does not use the return value of the first call to the function `assignMemory`, Polyspace reports a violation.
- Because the return value of the second call to `assignMemory` is assigned to a local variable, it is compliant with this rule.
- Because the third call to `assignMemory` is cast to `void`, it is compliant with this rule
- Because `main` does not use the object returned by the function `std::vector::erase()`, Polyspace reports a violation.

## Check Information

**Group:** Language Independent Issues
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A0-1-3

Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used

## Description

### Rule Definition

*Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used.*

### Rationale

Functions defined in an anonymous namespace and static functions with internal linkage are callable only inside the compilation unit in which they are defined. Similarly, private member functions are callable only inside the class implementation that they belong to. In both these cases, such functions are intended to be used exclusively in the current source code and not in external code that is integrated later on into the project. Not using such functions indicates poor software design or missing logic in the current code base.

**Note** An explicit function call in the source code is sufficient to satisfy this rule, even if the call is not reachable at run time. A separate rule, M0-1-1, checks for all unreachable code occurrences.

### Polyspace Implementation

If a function defined in your source code is not called explicitly and belongs to one of these categories, the checker flags the function definition:

- Functions defined in anonymous namespace
- Static functions with internal linkage
- Private member functions that are defined outside the class definition

The checker does not flag an uncalled private member function that is defined inside the class definition.

*The checker does not flag private member functions that are defined outside the class definition in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Function in Anonymous Namespace Not Used

```
#include <cstdint>
```

```
namespace
{
  void F1()   // Compliant, function in anonymous namespace used
  {
  }

  void F2()   // Noncompliant, function in anonymous namespace not used
  {
  }
}

int main()
{
  F1();
  return 0;
}
```

The static function F2 is defined in an anonymous namespace but is not called from the main function, thus violating this coding rule.

### Static Function Not Used

```
#include <cstdint>
static void F1() // Compliant, static function called from main
{
}

static void F2() // Noncompliant, static function not called from main
{
}

int main()
{
  F1();
  return 0;
}
```

The static function F2 has internal linkage but is not called from the main function, thus violating this coding rule.

### Private Member Function Not Used

```
#include <cstdint>

class C
{
  public:
    C() : x(0) {}
    void M1(std::int32_t);
    void M2(std::int32_t, std::int32_t);
  private:
    std::int32_t x;
    void M1PrivateImpl(std::int32_t j);
};

// Compliant, member function is used
void C::M1(std::int32_t i)
{
  x = i;
}

// Compliant, never used but declared as public
void C::M2(std::int32_t i, std::int32_t j)
{
  x = (i > j) ? i : j;
}

void C::M1PrivateImpl(std::int32_t j) // Noncompliant, private member function never used
{
  x = j;
}

int main()
{
  C c;
  c.M1(1);
  return 0;
}
```

The private member function `M1PrivateImpl` is not called from any member of the class `C`, thus violating this coding rule.

## Check Information

**Group:** Language independent issues
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A0-1-4

There shall be no unused named parameters in non-virtual functions

## Description

### Rule Definition

*There shall be no unused named parameters in non-virtual functions.*

### Rationale

Unused parameters can indicate that the code is possibly incomplete. The parameter is possibly intended for an operation that you forgot to code or leftover from a design change.

If the parameters are obtained by copy and the copied objects are large, the redundant copies can slow down performance.

### Polyspace Implementation

The checker flags a function that has unused named parameters unless the function body is empty.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Independent Issues
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A0-1-5

There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it

## Description

### Rule Definition

*There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.*

### Rationale

Unused parameters can indicate that the code is possibly incomplete. The parameter is possibly intended for an operation that you forgot to code.

The rule focuses on virtual functions because all functions that override a virtual function must have the same signature as the virtual function, including number and type of parameters. If a parameter is indeed not required, the issue can cascade from the original function to all overriding functions.

However, in an overriding function, you might not have need for a certain parameter. You can leave that parameter unnamed. This rule enforces the convention that unused parameters stay unnamed.

### Polyspace Implementation

For each virtual function, the checker looks at all overrides of the function. If an override has a named parameter that is not used, the checker shows a violation on the original virtual function and lists the override as a supporting event.

Note that Polyspace checks for unused parameters in virtual functions within single translation units. For instance, if a base class contains a virtual method with an unused parameter but the derived class implementation of the method uses that parameter, the rule is not violated. However, if the base class and derived class are defined in different files, the checker, which operates file by file, flags a violation of this rule on the base class.

The checker does not flag unused parameters in functions with empty bodies.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language independent issues
**Category:** Required, Automated

## Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A0-1-6

There should be no unused type declarations

## Description

### Rule Definition

*There should be no unused type declarations.*

### Rationale

If you declare a type but do not use it, determining if the type is redundant or left unused by mistake is difficult when you review your code.

For example, if you declare an enumerated data type to store some specialized data, but then use an integer type to store the data, the unused type can indicate a coding error.

### Polyspace Implementation

Polyspace reports a violation of this rule if your code contains unused local types.

As an exception, Polyspace does not report a violation if you define the unused type with `public` access in a template class.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unused `enum` Declaration

In this example, you do not use the enumerated type `switchValue`.Because you declare the type but do not use it, Polyspace reports a violation.

```
enum switchValue {low, medium, high}; //Noncompliant

void operate(int userInput) {
    switch(userInput) {
        case 0: // Turn on low setting
                 break;
        case 1: // Turn on medium setting
                 break;
        case 2: // Turn on high setting
                 break;
        default: // Return error
    }
}
```

Perhaps the intention was to use the type as `switch` input like this.

```
enum switchValue {low, medium, high}; //Compliant

void operate(switchValue userInput) {
    switch(userInput) {
        case low: // Turn on low setting
                   break;
        case medium: // Turn on medium setting
                     break;
        case high: // Turn on high setting
                   break;
        default: // Return error
    }
}
```

#### Public Unused Types in Templates

When developing template classes, you might declare a public `typedef` that is used by the clients of the template, even if the template itself does not use the `typedef`. In such a case, Polyspace does not report a violation if a public `typedef` remains unused in a template. For instance, in this code, Polyspace does not report a violation for the unused type `T*`.

```
template<typename T>
  class myContainerIterator {
    public:
      using pointer = T*;              // Compliant
      typedef T* Pointer;              // Compliant

    //...
  };
```

### Check Information

**Group:** Language Independent Issues
**Category:** Advisory, Automated

## Version History

**Introduced in R2019a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A0-4-2

Type long double shall not be used

## Description

### Rule Definition

*Type long double shall not be used.*

### Rationale

The size of `long double` is implementation-dependent and reduces the portability of your code across compilers. Compilers can implement `long double` as a synonym for `double` or an 80-bit extended precision type or 128-bit quadruple precision type that are more precise than `double`.

Instead, for multiple precision arithmetic that requires types more precise than `double`, use libraries that support multiple precision arithmetic with well-defined data types.

### Polyspace Implementation

The rule checker flags all uses of the `long double` keyword.

If you do not want to fix the issue, add a comment justifying the result. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use of `long double` Keyword

```
void func() {
  float f{0.1F};  //Compliant
  double D(0.1);  //Compliant
  long double LD(0.1L);  //Noncompliant
}
```

The use of `long double` violates this rule.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A0-4-4

Range, domain and pole errors shall be checked when using math functions

## Description

### Rule Definition

*Range, domain and pole errors shall be checked when using math functions.*

### Rationale

Before using a math function, check input arguments for potential domain, range, and pole errors.

This checker searches for domain errors when a function argument falls outside the allowed domain, pole errors when finite arguments result in infinite results, and range errors when results of a function cannot be represented by the return value limitations.

Domain, pole and range errors result in unexpected or undefined behavior.

### Polyspace Implementation

Polyspace raises this defect when you call a math function that results in a domain, pole, or range error.

By default, a Bug Finder analysis does not recognize infinities and `NaNs`. Operations that results in infinities and `NaNs` might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats)`.

### Extend Checker

Extend this checker to check for defects caused by specific values and invalid use of functions from a custom library. For instance:

- You might be using a custom library of mathematical functions. If a custom library function have the same domain and range as another function from the standard library, you can extend this checker to check the custom library function. See “Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries”.
- A default Bug Finder analysis might not raise a defect when the input values are unknown and only a subset of inputs can cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Check for Range, Domain, and Pole Errors Before Using Math Functions

```
#include <cmath>

double getSqrt(double val)
{
    return sqrt(val);                       //Noncompliant
}

double getRefinedSqrt(double val)
{
    if (val >= 0)
        return sqrt(val);                   //Compliant
    else
        return 0;
}


void main()
{
    double root, refinedRoot;

    root = getSqrt(4);
    root = getSqrt(-1);

    refinedRoot = getRefinedSqrt(4);
    refinedRoot = getRefinedSqrt(-1);
}
```

Because the math function `sqrt` within the function `getSqrt` results in a domain error, Polyspace flags it as noncompliant. Performing a check on the variable `val`, as seen with the function `getRefinedSqrt`, helps ensure the value passed to the math function `sqrt` is compliant and expected.

## Check Information

**Group:** Language independent issues
**Category:** Required, Partially automated

# Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A1-1-1

All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features

## Description

### Rule Definition

*All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features.*

### Polyspace Implementation

The checker reports compilation errors as detected by a compiler that strictly adheres to the C++03 Standard (ISO/IEC 14882:2003).

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** General
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A2-3-1

Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code

## Description

### Rule Definition

*Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.*

### Rationale

In the C++ standard, the basic source character set consists of 96 characters. They are:

- The space character.
- The control characters such as horizontal tab, vertical tab, form feed, and new line.
- Upper and lower case letters, and numbers.
- Special characters, such as _ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '.

Using characters outside this set can cause confusion and unexpected bugs. For example, the Greek letter "Τ" is visually similar to the English letter "T", but they are separate characters with different unicode code-point values. To avoid unexpected behavior, use only the above specified characters in your source code, including comments and string literals. You can use characters outside this set in only two cases. You can use:

- Other characters inside the text of a wide string or a UTF-8 encoded string.
- The character @ inside comments, the text of a wide string, or a UTF-8 encoded string.

### Polyspace Implementation

Polyspace flags the characters in your source code that are not in the set of 96 characters specified in C++ standard, with two exceptions that come from the AUTOSAR C++14 Standard. Polyspace does not flag:

- Other characters inside the text of a wide string or a UTF-8 encoded string.
- The character @ inside comments, the text of a wide string, or a UTF-8 encoded string.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Do Not Use Characters Outside the Specified Set

The following example demonstrates the Polyspace implementation of AUTOSAR rule A2-3-1.

```
#include <cstdint>

// @ brief foo function    //Compliant by exception
/* @ brief foo function */ //Compliant by exception

#if 0
@ This one is not in a comment  //Noncompliant
#endif
/*Define £ and € as currency */   // Noncompliant
#define CUR1 "£"   //Noncompliant
#define CUR2 "€"   //Noncompliant
void myfunction(char *str );
int  Total = 0;    //Complaint
int  Total = 0;    //Noncompliant
void foo()
{
    char *s1 = "Greek T  - normal string"; //Noncompliant
    wchar_t *s2 { L"Greek T @ wide string"};  //Compliant
    char *s3 = u8"Greek T @ UTF-8";          // Compliant
    char16_t *s4 = u"Greek T UTF-16";  //Noncompliant
    char32_t *s5 = U"Greek T UTF-32";  //Noncompliant
    char *s6 = "mail@company.com"; //Noncompliant
    myfunction("Greek T");//Noncompliant
    myfunction(s3);//Complaint
}

main(){
    // ..
}
```

If your code has characters that are not in the specified character set, Polyspace flags them. Note the global variables `Total` and `Total`. Even though it looks as if they are the same variable, they are two different variables because the latter starts with the Greek letter "T". Confusion between these two characters can lead to unexpected behavior. Because the Greek letter "T" is outside the standard set of characters, Polyspace flags every use of the character, even those in comments and string literals.

Polyspace flags every use of characters outside the specified set, with the following exceptions. You can use:

- Other characters inside a wide string such as `s2` or UTF-8 encoded string such as `s3`.
- The character `@` inside a wide string such as `s2`, a UTF-8 encoded string such as `s3`, or a comment.

## Check Information

**Group:** Lexical conventions
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**
"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A2-5-1

Trigraphs shall not be used

## Description

### Rule Definition

*Trigraphs shall not be used.*

### Rationale

You denote trigraphs with two question marks followed by a specific third character (for instance,`'??-'` represents a `'~'` (tilde) character and `'??)'` represents a `']'`). These trigraphs can cause accidental confusion with other uses of two question marks.

For instance, the string

```
"(Date should be in the form ??-??-??)"
```

is transformed to

```
"(Date should be in the form ~~]"
```

but this transformation might not be intended.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Lexical Conventions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A2-5-2

Digraphs shall not be used

## Description

### Rule Definition

*Digraphs shall not be used.*

### Rationale

Digraphs are a sequence of two characters that are supposed to be treated as a single character. The checker flags use of these digraphs:

- `<%`, indicating {
- `%>`, indicating }
- `<:`, indicating [
- `:>`, indicating ]
- `%:`, indicating #
- `%:%:`, indicating ##

When developing or reviewing code with digraphs, the developer or reviewer can incorrectly consider the digraph as a sequence of separate characters.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Lexical Conventions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A2-7-1

The character \ shall not occur as a last character of a C++ comment

## Description

### Rule Definition

*The character \ shall not occur as a last character of a C++ comment.*

### Rationale

If your code has the character \ at the end of a single-line comment, the next line of code becomes a continuation of the comment. Ending single line comments by using the character \ can inadvertently comment-out sections of code.

### Polyspace Implementation

Polyspace checks if the character \ is the last character of a C++ comment .

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using the character \ as Last Character of C++ Comments

```
#include <cstdint>

void foo()
{
  int32_t idx = 0;
  int32_t limit = 20;
  int32_t count = 20;
  ++idx; // Incrementing index before the loop starts// Requirement X\\
  for(;idx<limit;++idx)
  {
     --count;
  }
}
```

The `for` loop definition is commented-out because the single-line comment ends with the character \. As a result, `count` is decremented only once, perhaps inadvertently. The checker flags this issue by highlighting the character \ in the single-line comment.

## Check Information

**Group:** Lexical conventions
**Category:** Required, Automated

## Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A2-7-2

Sections of code shall not be "commented out"

## Description

### Rule Definition

*Sections of code shall not be "commented out".*

### Rationale

Commenting out code is not a good practice. The commented out code can remain out of sync with the surrounding code without causing compilation errors. Later, if you uncomment the code, you can encounter unexpected issues.

In addition, C-style comments enclosed in `/* */` do not support nesting. A comment beginning with `/*` ends at the first `*/` even when the `*/` is intended as the end of a later nested comment. If a section of code that is commented out already contains comments, you can encounter compilation errors (or at least comment out less code than you intend).

Use comments only to explain aspects of the code that are not apparent from the code itself.

### Polyspace Implementation

The checker uses internal heuristics to detect commented out code. For instance, characters such as `#`, `;`, `{` or `}` indicate comments that might potentially contain code. These comments are then evaluated against other metrics to determine the likelihood of code masquerading as comment. For instance, several successive words without a symbol in between reduces this likelihood.

The checker does not flag the following comments even if they contain code:

- Doxygen comments beginning with `/**`, `/*!`, `///` or `//!`.
- Comments that repeat the same symbol several times, for instance, the symbol = here:

  ```
  // =====================================
  // A comment
  // =====================================*/
  ```
- Comments on the first line of a file.
- Comments that mix the C style (`/* */`) and C++ style (`//`).

The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Code Commented Out With C-Style Comments

```
#include <iostream>
/* class randInt {
    public:
      int getRandInt();
};
*/

int getRandInt();

/* Function to print random integers*/
void printInteger() {
    /* int val = getRandInt();
     * val++;
     * std::cout << val;*/
    std::cout << getRandInt();
}
```

This example contains two blocks of commented out code, that constitutes two rule violations.

### Code Commented Out With C++-Style Comments

```
#include <iostream>
int getRandInt();

// Function to print random integers
void printInteger() {
    // int val = getRandInt();
    // val++;
    // std::cout << val;
    std::cout << getRandInt();
}
```

This example contains a block of commented out code that violates the rule.

## Check Information

**Group:** Lexical Conventions
**Category:** Required, Non-automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A2-7-3

All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation

## Description

### Rule Definition

*All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation.*

### Rationale

This rule requires developers to document externally visible declarations so that users of the declared types and functions can form expectations based on this documentation.

In comments preceding the declarations, developers can document information such as function and method usage, parameter descriptions, exceptions thrown, and other specifications such as side effects, memory management and ownership.

### Polyspace Implementation

In cases where a declaration comes before a definition, the checker flags the declaration if there are no preceding comments. Otherwise, the checker flags the definition.

There can be at most one blank line between a declaration or definition and the preceding comment.

In some cases, you might want to disable the rule or justify some violations. For instance:

- Legacy projects might contain many insufficiently documented types or functions. Unless you want to clean up these projects, you might consider disabling this rule.
- In code documentation tools such as Doxygen, you can add documentation comments *after* a data member or member function. In Doxygen, if you begin the comment with <, the tool considers the comment as documentation for the data member or member function. For instance:

  ```
  int var; /*!< Data member description*/
  ```

  However, Polyspace considers such declarations or definitions as rule violations. If you want to continue using this style of documentation comments, you might consider justifying the violations.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant and compliant class definitions

```
#include <cstdint>
```

```
class aClass { //Noncompliant class definition
public:
    aClass(std::int32_t aParameter): aVar(aParameter) {} //Noncompliant
private:
    std::int32_t aVar; //Noncompliant variable definition
};

/// @desc Class responsibilities

class anotherClass { //Compliant class definition
public:
    /// @desc Constructor description
    ///
    /// @param aParameter Parameter description
    anotherClass(std::int32_t aParameter): anotherVar(aParameter) {} //Compliant
private:
    /// @desc Data member description
    std::int32_t anotherVar; //Compliant variable definition
};
```

In this example, the definition of class `aClass` has three rule violations. The class definition itself, the constructor definition, and the definition of data member `aVar` are all missing preceding comments explaining the definitions.

The class `anotherClass` is a compliant version of the same class that satisfies the requirements of this rule.

## Check Information

**Group:** Lexical conventions
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A2-8-1

A header file name should reflect the logical entity for which it provides declarations.

## Description

### Rule Definition

*A header file name should reflect the logical entity for which it provides declarations.*

### Rationale

A header file name that matches the name of the entity that is declared in that file makes your `#include` directives clearer and your code more readable.

### Polyspace Implementation

Polyspace checks the header file name against the name of relevant declared types such as `class` or `struct`, or `namespace` names. If the names do not match, Polyspace flags the first character on the first line of the header file.

- The name comparison is case insensitive. For instance, `myheader` matches `myHeader`.
- The name comparison ignores:
  - The underscore character '`_`'. For instance, `myheader` matches `my_Header`.
  - Prefix characters '`C`', '`M`', '`T`', or suffix character '`T`'. The comparison ignores either the prefix or suffix characters, but not both. For instance, `myheader` matches `CmyHeader` and `myHeader_T`, but not `CmyHeader_T`.
  - The hyphen character '`-`' in file names. For instance, a file named `my-header.h` matches a `struct` named `_myHeader`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Header File Name Does Not Match `class` Name

**myheader.h**

```
#include <memory> //Noncompliant - header name does not match the class name
#include <string>

class myClass
{
    virtual ~myClass()=default;
};

class Player : public myClass
{
    std::string Name;
    int Rank;
};
```

**file.cpp**

```
#include "myheader.h"

int main(){
  return 0;
}
```

In this example, the name of header file `myheader.h` is not compliant because it does not match the name of the base class (`myClass`) declared in that header file .

## Check Information

**Group:** Lexical conventions
**Category:** Required, Non-automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A2-8-2

An implementation file name should reflect the logical entity for which it provides definitions.

## Description

### Rule Definition

*An implementation file name should reflect the logical entity for which it provides definitions.*

### Rationale

An implementation file name that matches the name of the entity that is defined in that file makes your project structure clearer and your code more readable.

### Polyspace Implementation

Polyspace checks the implementation file name against the name of relevant defined types such as `class` or `struct`, or `namespace` names. If the names do not match, Polyspace flags the first character on the first line of the implementation file.

- The name comparison is case insensitive. For instance, `myclass` matches `myClass`.
- The name comparison ignores:
  - The underscore character '`_`'. For instance, `myclass` matches `my_Class`.
  - Prefix characters '`C`', '`M`', '`T`', or suffix character '`T`'. The comparison ignores either the prefix or suffix characters, but not both. For instance, `myclass` matches `CmyClass` and `myClass_T`, but not `CmyClass_T`.
  - The hyphen character '`-`' in file names. For instance, a file named `my-class.cpp` matches a `class` named `myClass_`.

Polyspace does not check the file where you implement `main()`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Implementation File Name Does Not Match `class` Name

**username.h**

```
#include <string>

class User
{
public:
    User();
    User(std::string s);
    std::string getUser();
private:
    std::string user;
};
```

**username.cpp**

```
#include "username.h" // Non-compliant

User::User() : user("") { }
User::User(std::string s): user(s) {}
std::string User::getUser()
{
    return user;
}
```

In the is example, the name of implementation file `username.cpp` is not compliant because it does not match the name of the class (`User`) defined in that file.

## Check Information

**Group:** Lexical conventions
**Category:** Advisory, Non-automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A2-10-1

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

## Description

### Rule Definition

*An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.*

### Rationale

The rule flags situations where the same identifier name is used in two variable declarations, one in an outer scope and the other in an inner scope.

```
int var;
...
{
...
  int var;
...
}
```

All uses of the name in the inner scope refers to the variable declared in the inner scope. However, a developer or code reviewer can incorrectly assume that the usage refers to the variable declared in the outer scope. In all cases flagged by this rule, you cannot clarify the usage further using the scope resolution operator.

### Polyspace Implementation

The rule checker flags all cases of variable shadowing except when:

- The same identifier name is used in an outer and inner named namespace.
- The same name is used for a class data member and a variable outside the class.
- The same name is used for a method in a base and derived class.

The checker flags even those cases where the variable declaration in the outer scope occurs *after* the variable declaration in the inner scope. In those cases, though the variable hiding does not occur, reusing the variable name can cause developer confusion.

The rule does not flag these situations because you can clarify whether an usage of the variable refers to the variable in the inner or outer scope. For instance, in this example:

```
int var;

namespace n1 {
   int var;
}
```

within the namespace `n1`, you can refer to the variable in the inner scope as `n1::var` and the global variable as `::var`.

The rule checker also does not detect these issues:

- A variable in an unnamed namespace hides another variable in an outer scope.
- A variable local to a lambda expression hides a captured variable.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Local Variable Hiding Global Variable

```
int varInit = 1;

void doSomething(void);

void step(void) {
    int varInit = 0; //Noncompliant
    if(varInit)
       doSomething();
}
```

In this example, `varInit` defined in `func` hides the global variable `varInit`. The `if` condition refers to the local `varInit` and the block is unreachable, but you might expect otherwise.

### Loop Index Hiding Variable Outside Loop

```
void runSomeCheck(int);

void checkMatrix(int dim1, int dim2) {
  for(int index = 0; index < dim1; index++) {
      for(int index = 0; index < dim2; index++) { // Noncompliant
          runSomeCheck(index);
      }
  }
}
```

In this example, the variable `index` defined in the inner `for` loop hides the variable with the same name in the outer loop.

## Check Information

**Group:** Identifiers
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A2-10-4

The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace

## Description

### Rule Definition

*The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace.*

### Rationale

You use namespaces to narrow the scope of the identifiers that you declare within these namespaces. This prevents these identifiers from being mistaken with identical identifiers in other scopes. If you reuse an identifier with static storage duration within the same namespace across source files, you might mistake one identifier for the other.

### Polyspace Implementation

- When you reuse identifiers, Polyspace flags the last use of the identifier if they are in the same translation unit. If the identifiers are in separate files, the identifier in the last file path by alphabetical order is flagged.

  However, if you reuse an identifier but declare only one instance of the identifier with the keyword `static`, that identifier is flagged regardless of the order in which the identifiers are declared.
- Polyspace raises no violation if you declare an identifier in a namespace and you reuse that identifier in the same namespace, but within a nested or inlined namespace. For instance, no violation is raised on `reusedVar` in this code snippet.

  ```
  //file1.cpp
  namespace foo {
    static int reusedVar; //resuedVar has static storage duration
  }

  //file2.cpp
   namespace foo {
    void func();
    namespace nested_foo {
      float reusedVar;
    }
    inline namespace inlined_foo {
      char reusedVar;
    }
  }
  ```

The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` or `extern` functions

- Uncalled and undefined local functions
- Unused types and variables

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Reuse of identifier within a namespace

**file1.cpp**

```
#include <cstdint>

namespace first_namespace
{
    static std::int32_t global_var; //Compliant- Reused in global namespace
    static std::int32_t reusedVar1;  //Noncompliant
    void reusedVar2();
    static std::int32_t reusedVar3; //Noncompliant
    void use(){
        ++reusedVar1;
        reusedVar2();
        ++reusedVar3;
    }
}
static std::int32_t file_var = 10; //Compliant - identifier not reused
```

**file2.cpp**

```
#include <cstdint>



static std::int32_t global_var; //Compliant - Reused in global namespace



namespace first_namespace
{
    std::int32_t reusedVar1;
    static std::int32_t reusedVar2;  //Noncompliant
    void f()
    {
        float reusedVar3;
        ++reusedVar3;
    }
    void otherUse() {
        ++reusedVar2;
    }
}



namespace second_namespace
{
    std::int32_t reusedVar1; //Compliant - Reused in different namespace
}
```

In this example, Polyspace flags the reuse of `resusedVar1`, `reusedVar2`, and `reusedVar3` in the same namespace in both files. Polyspace does not flag the reuse of `reusedVar1` in a different namespace in `file2.cpp`. Note that when only one instance of the reused identifier is declared with the keyword `static`, Polyspace flags that instance. The identifier `global_var` is not flagged because it is declared in different namespaces, the global namespace, and `first_namespace`.

## Check Information

**Group:** Lexical conventions
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A2-10-5

An identifier name of a function with static storage duration or a non-member object with external or internal linkage should not be reused

## Description

### Rule Definition

*An identifier name of a function with static storage duration or a non-member object with external or internal linkage should not be reused.*

### Rationale

Objects with static storage duration remain available during the entire execution of the program. These include:

- Non-member objects with external linkage that can be referred to from any of the translation units of your project.
- Objects declared with the `static` class specifier. These objects have internal linkage and can be referred to from any scope within their translation unit.

If you reuse the name of an identifier, you might mistake one identifier for the other.

The rule does not apply to objects with no linkage, for instance function local static objects, since the identifiers of those objects cannot be referred to from outside of their scope.

### Polyspace Implementation

- When you reuse identifiers, Polyspace flags the last use of the identifier if they are in the same translation unit. If the identifiers are in separate files, the identifier in the last file path by alphabetical order is flagged.
- If you declare a function in a namespace with the `static` class specifier and reuse the function identifier to declare a non-static function in another namespace, Polyspace flags the identifier of the static function. For instance, in this code snippet, the identifier `func` is reused in namespace `NS_2` but it is flagged in namespace `NS_1`.

```
namespace NS_1 {
    static void func(void); // Polyspace flags this use of "func".
};

namespace NS_2 {
    void func(void); //"func" identifier reused but this is not a static function.
}
```

- Polyspace flags the identifier of a global variable if you reuse the identifier for a local variable.
- Polyspace does not flag the reuse of an identifier for global functions and their arguments that are declared without the `static` class specifier.

The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` or `extern` functions

- Uncalled and undefined local functions
- Unused types and variables

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Reuse of identifier with static storage duration

### file1.cpp

```
#include <cstdint>

namespace first_namespace
{
    static std::int32_t global_var = 0;

}
static std::int32_t file_var = 10; //Compliant - identifier not reused
```

### file2.cpp

```
;

#include <cstdint>

namespace first_namespace
{
    static std::int32_t global_var = 0;  // Noncompliant - identifier reused
    static std::int16_t module_var = 20; // Compliant - identifier not reused
}




namespace second_namespace
{

    void globalfunc(int argument) // non-static global function and arguments do not raise violation
    {
        int local_var; // local variable
        static std::int16_t local_static; // Object with no linkage
    }
    std::int16_t globalvar_reusedinlocal;
    std::int16_t globalvar_notreused; // Compliant, identifier not reused
    void foo(){
        ++globalvar_reusedinlocal;
        ++globalvar_notreused;
    }
};

namespace third_namespace
{

    void globalfunc(int argument) // non-static global function and arguments do not raise violation
    {
        static std::int16_t local_static; // Object with no linkage
        int local_var; // local variable
        int globalvar_reusedinlocal; // Non-compliant, identifier reused in local variable
        ++globalvar_reusedinlocal;
    }

};
```

In this example, `global_var` is declared with the `static` class specifier in source file file1.cpp. This identifier is reused in source file file2.cpp. In the same file, `globalvar_reusedinlocal` is declared in `second_namespace` and has external linkage. This declaration is non-compliant because the identifier is reused for the local variable in `globalfunc`.

## Check Information

**Group:** Lexical conventions
**Category:** Advisory, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A2-10-6

A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope

## Description

### Rule Definition

*A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope.*

### Rationale

When a variable, data member, function, or enumerator shares its name with a class or enumeration in the same scope, the latter is hidden. That is, all uses of the name refers to the variable, data member, function, or enumerator instead of the class or enumeration, regardless of declaration order. Hidden classes or enumerations can be misleading and can lead to compilation errors. Do not re-use names to declare classes and enumerations.

### Polyspace Implementation

Polyspace flags the declaration of a variable, data member, function, or enumerator that shares the name of a class or enumeration in the same block.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Do Not Hide Class Declarations

The following example demonstrate the Polyspace implementation of AUTOSAR rule A2-10-6.

```
#include <cstdint>
namespace NS1
{
    class G {};
    void G() {}             //Noncompliant
}
namespace NS2
{
    enum class H { VALUE=0, };
    std::uint8_t H = 17;    //Noncompliant
}
namespace NS3
{
    class J {};
    enum H {
        J=0,                 // Noncompliant
    };
}
main()
{
    //...
}
```

Polyspace flags the declaration of the:

- Function `G()` because it hides the class `G` declared in the same block.
- Variable `H` because it hides the enumeration `H` declared in the same block.
- Enumerator `J` because it hides the class `J` is declared in the same block.

## Check Information

**Group:** Lexical conventions
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A2-11-1

Volatile keyword shall not be used

## Description

### Rule Definition

*Volatile keyword shall not be used.*

### Polyspace Implementation

Reports if volatile keyword is used.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Lexical Conventions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A2-13-1

Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used

## Description

### Rule Definition

*Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used.*

### Rationale

Escape sequences are certain special characters represented in string and character literals. They are written with a backslash (\) followed by a character.

The C++ Standard (ISO/IEC 14882:2003, Sec. 2.13.2) defines a list of escape sequences. See Escape Sequences. Use of escape sequences (backslash followed by character) outside that list leads to undefined behavior.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unrecognized Escape Sequences

```
#include <string>
void func () {
  const std::string a = "\k"; //Noncompliant
  const std::string b = "\U0001f34c"; //Compliant
}
```

In this example, Polyspace reports a violation on the unrecognized escape sequence `\k`.

## Check Information

**Group:** Lexical Conventions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A2-13-2

String literals with different encoding prefixes shall not be concatenated

## Description

### Rule Definition

*String literals with different encoding prefixes shall not be concatenated.*

### Rationale

Narrow string literals are enclosed in double quotes without a prefix. Wide string literals are enclosed in double quotes with a prefix L outside the quotes. See string literals.

Concatenation of narrow and wide string literals can lead to undefined behavior.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Concatenation of Narrow and Wide String Literals

```
char array[] = "Hello" "World";
wchar_t w_array[] = L"Hello" L"World";
wchar_t mixed[] = "Hello" L"World"; //Noncompliant
```

In this example, in the initialization of the array `mixed`, the narrow string literal `"Hello"` is concatenated with the wide string literal `L"World"`.

## Check Information

**Group:** Lexical Conventions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A2-13-3

Type wchar_t shall not be used

## Description

### Rule Definition

*Type wchar_t shall not be used.*

### Rationale

The size of `wchar_t` is implementation-dependent. If you use `wchar_t` for Unicode values, your code is bound to a specific compiler.

To improve the portability of your code, use `char16_t` and `char32_t` instead. These are standard types introduced in C++11 for text strings with UTF-16 and UTF-32 encodings.

### Polyspace Implementation

The rule checker flags all uses of the `wchar_t` keyword.

If you do not want to fix the issue, add a comment justifying the result. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use of `wchar_t` Keyword

```
char16_t str1[] = u"A UTF-16 string";  //Compliant
char32_t str2[] = U"A UTF-32 string";  //Compliant
wchar_t str3[] = L"A Unicode string";  //Noncompliant
```

The use of `wchar_t` violates this rule. Instead the types `char16_t` and `char32_t` can be used.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A2-13-4

String literals shall not be assigned to non-constant pointers

## Description

### Rule Definition

*String literals shall not be assigned to non-constant pointers.*

### Rationale

This rule prevents assignments of string literals to pointers that point to non `const` objects. Such assignments allow later modification of the string literal.

An attempt to modify a string literal can result in undefined behavior. For example, some implementations can store string literals in read-only memory. An attempt to modify the string literal can result in an exception or crash.

Later C++ standards require a compiler warning for such modifications. The rule is in place for situations when you suppress compiler warnings (and AUTOSAR C++14 rules associated with those warnings).

### Polyspace Implementation

The rule checker flags assignment of string literals to pointers other than pointers to `const` objects.

The checker does not flag assignment of string literals to non-`const` arrays. The checker for `AUTOSAR C++ 14 Rule A18-1-1` forbids direct use of C-style arrays and prevents these assignments.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Incorrect Assignment of String Literal

```
char *str1 = "xxxxxx";            // Non-Compliant
const char *str2 = "xxxxxx";      // Compliant

void checkSystem1(char*);
void checkSystem2(const char*);

void main() {
 checkSystem1("xxxxxx");    // Non-Compliant
 checkSystem2("xxxxxx");    // Compliant
}
```

In this example, the rule is not violated when string literals are assigned to `const char*` pointers, either directly or through copy of function arguments. The rule is violated only when the `const` qualifier is not used.

### Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A2-13-5

Hexadecimal constants should be uppercase

## Description

### Rule Definition

*Hexadecimal constants should be uppercase.*

### Rationale

Hexadecimal notation uses digits 0–9 and letters A to F. Using only uppercase alphabetic letters in a hexadecimal constant definition helps keep the source code consistent, readable, and easy to maintain.

A mix of uppercase and lowercase hexadecimal constants might lead to confusion in the development process, which in turn might lead to bugs. Consistently enforcing the exclusive use of uppercase hexadecimal constants reduces this potential issue.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Lowercase Alphabetic Letters in Hexadecimal Constant Definitions

This example shows the Polyspace implementation of AUTOSAR rule A2-13-5.

```
#include <cstdint>

int main(void)
{
  std::int16_t a = 0x0f0f; //Noncompliant
  std::int16_t b = 0x0f0F; //Noncompliant
  std::int16_t c = 0x0F0F; //Compliant
  return 0;
}
```

In this example, three hexadecimal constants are defined. All three constants have the same value, but two of these definitions use lowercase letters. Because the definitions of the hexadecimal constants a and b do not use uppercase letters exclusively, Polyspace flags their definitions as noncompliant with this rule. Because the definition of the hexadecimal constant c uses uppercase letters exclusively, Polyspace does not flag its definition as noncompliant with this rule.

## Check Information

**Group:** Lexical Conventions
**Category:** Advisory, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A2-13-6

Universal character names shall be used only inside character or string literals

## Description

### Rule Definition

*Universal character names shall be used only inside character or string literals.*

### Rationale

Universal character names are a way to represent unicode characters by using code points. For example, `\U0000231A` represents the unicode character '⌚'. When you use universal character names to define an identifier, it is difficult to read the source code. Using universal character names as identifier is confusing and troublesome. Avoid using universal character names outside a character or string literal.

### Polyspace Implementation

Polyspace flags the use of universal character names outside a character or string literal.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Do Not Use Universal Character Names Outside Character or String Literal

The following example demonstrate the Polyspace implementation of AUTOSAR rule A2-13-6.

```
#include <cstdint>
#define \U0000231AMACRO(x) (x)     // Noncompliant
void €uro(){            // Compliant
    std::int32_t €uro;    // Compliant
    std::int32_t \U0000231Ahello; // Noncompliant
    wchar_t wc = '\U0000231A';    // Compliant
    std::int32_t Hello\U0000231AWorld;  // Noncompliant
}
typedef struct \U0000231Astruct {    // Noncompliant
    std::int32_t regular;
    std::int32_t €uro;                // Compliant
    std::int32_t \U0000231Ahello;     // Noncompliant
} \U0001f615type;                // Noncompliant

main(){
    //...
}
```

The variable `\U0000231Ahello` is declared using universal character name. Such a variable name is difficult to use, and makes the code confusing. Polyspace flags the use of universal character names outside a character or string literal.

## Check Information

**Group:** Lexical conventions
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A3-1-1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule

## Description

### Rule Definition

*It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.*

### Rationale

If a header file with variable or function definitions appears in multiple inclusion paths, the header file violates the One Definition Rule possibly leading to unpredictable behavior. For instance, a source file includes the header file `include.h` and another header file, which also includes `include.h`.

### Polyspace Implementation

The rule checker flags variable and function definitions in header files.

Polyspace reports violation of this rule in header files. In a nonheader source file, violation of this rule is not reported.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Variable and Function Definitions in Header Files

In this example, Polyspace reports violations of this rule when variables and function definitions are placed in the header file `header.hpp`.

```
// header.hpp
#include <cstdint>
void F1();         // Compliant
extern void F2(); // Compliant
void F3()         // Noncompliant
{
}
static inline void F4()
{
} // Compliant
template <typename T>
void F5(T) // Compliant
{
}
std::int32_t a;                        // Noncompliant
extern std::int32_t b;                 // Compliant
constexpr static std::int32_t c = 10; // Compliant
namespace ns
{
    constexpr static std::int32_t d = 100; // Compliant
    const static std::int32_t e = 50;      // Compliant
    static std::int32_t f;                 // Noncompliant
    static void F6() noexcept;             // Noncompliant
}
```

```
//source.cpp
#include "header.hpp"
void foo()
{
    //...
}
```

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A3-1-2

Header files, that are defined locally in the project, shall have a file name extension of one of: `.h`, `.hpp` or `.hxx`

## Description

### Rule Definition

*Header files, that are defined locally in the project, shall have a file name extension of one of: `.h`, `.hpp` or `.hxx`.*

### Rationale

Developers and code reviewers expect a header file to have one of the standard file name extensions.

### Polyspace Implementation

The rule checker flags files included with the `#include` directive with names that have an extension other than `.h`, `.hpp` or `.hxx`. For instance:

```
#include <header.c>
#include <header2.cpp>
```

Instead of `<...>`, if you use `"..."` around the file, the checker also flags the case where the file does not have an extension at all.

The checker does not flag the following inclusions:

- Files included with the `Include (-include)` option.
- Included files that do not exist.

The checker is case-insensitive.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

# Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**
"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A3-1-3

Implementation files, that are defined locally in the project, should have a file name extension of ".cpp"

## Description

### Rule Definition

*Implementation files, that are defined locally in the project, should have a file name extension of ".cpp".*

### Polyspace Implementation

Not case sensitive if you set the option `-dos`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Basic Concepts
**Category:** Advisory, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A3-1-4

When an array with external linkage is declared, its size shall be stated explicitly

## Description

### Rule Definition

*When an array with external linkage is declared, its size shall be stated explicitly.*

### Rationale

Though you can declare an incomplete array type and later complete the type, specifying the array size during the first declaration makes the subsequent array access less error-prone.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Array Size Unspecified During Declaration

```
int array[10];
extern int array2[]; //Noncompliant
int array3[]= {0,1,2};
extern int array4[10];
```

In the declaration of `array2`, the array size is unspecified.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A3-1-5

A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template

## Description

### Rule Definition

*A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template.*

### Rationale

Placing a function definition in a class definition is allowed only if:

- The function is intended to be inlined. Placing the definition of a member function in the class definition instructs the compiler to inline the member function. Inlining small functions avoids the run-time overhead of function calls and improves the performance of the compiled executable. But if you place the definition of a large member function inside the class definition unaware of this implicit inlining, the compiled executable might be too large.
- The function is a member function template or a member of a class template. These coding practices reduce repetitions of template syntax elements (for example, the parameter list). This reduction improves the readability and maintainability of the code.

### Polyspace Implementation

The checker uses the heuristic that, unless you explicitly use the `inline` keyword, you intend to inline only small functions that consist of no more than one statement. The checker interprets AUTOSAR C++14 Rule A3-1-5 in the following way.

For nontemplate member functions and member functions of nontemplate classes, the checker flags one-line member functions defined outside a class and larger member functions defined inside a class.

For template member functions and member functions of template classes, the checker flags any member function that is defined outside a class.

Polyspace does not report a violation of this rule for `constexpr` and `consteval` functions.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Defining a Small Member Function Outside Class Definition

```
#include <cstdint>
#include <iostream>
```

```
class A
{
    private:
    std::uint32_t val = 5;

    public:
    std::uint32_t Foo()  // Compliant with (1)
    {
        return val;
    }

    std::uint32_t Bar();
};

std::uint32_t A::Bar() // Noncompliant with (1)
{
    return (val + 5);
}

std::uint32_t main()
{
    A a;
    std::cout << a.Foo() << std::endl;
    std::cout << a.Bar() << std::endl;
    return 0;
}
```

The placement of the definition of `Bar` outside the definition of class `A` violates the rule, because `Bar` consists of a single statement.

### Defining a Member Function Template Outside Class Definition

```
#include <cstdint>
#include <iostream>

class A
{
  public:
    template <typename T>     // Compliant with (2)
    void Foo(T t)
    {
      std::cout << "This function is defined inside with param: "
      << t << std::endl;
    }

    template <typename T>     // Non-compliant with (2)
    void Bar(T t);
};


template <typename T>
void A::Bar(T t)
{
  std::cout << "This function is defined outside with param: "
  << t << std::endl;
}

std::uint32_t main(void)
```

```
{
  A a;
  a.Foo<float>(3.14f);
  a.Bar<std::uint32_t>(5);
  return 0;
}
```

The placement of the definition of the member function template `Bar` outside the definition of class `A` violates the rule.

### Defining a Member Function Outside Class Template Definition

```
#include <cstdint>
#include <iostream>

template <typename T>
class B
{
  public:
    B(const T x) : t(x) {}

    void display()   //Compliant with (3)
    {
      std::cout << t << std::endl;
    }

    void display2();   //Non-compliant with (3)

  private:
    T t;
};

template <typename T>
void B<T>::display2()
{
  std::cout << t << std::endl;
}

int main(void)
{
  B<std::int32_t> b(7);
  b.display();
  b.display2();
  return 0;
}
```

The placement of the definition of the member function `display2` outside the definition of the class template `B` violates the rule.

## Check Information

**Group:** Basic concepts
**Category:** Required, Partially automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A3-1-6

Trivial accessor and mutator functions should be inlined

## Description

### Rule Definition

*Trivial accessor and mutator functions should be inlined.*

### Rationale

Inlined functions avoid the run-time overhead of function calls but can result in code bloat. If an accessor (getter) or mutator (setter) method is trivial, code bloat is not an issue. You can inline these methods to avoid the unnecessary overhead of function calls. You can also avoid repeating several syntax elements inside and outside the class definition.

Methods defined inside classes are implicitly considered as inlined methods. You can inline methods defined outside classes explicitly by using the `inline` keyword.

### Polyspace Implementation

To determine if a method is trivial, the checker uses this criteria:

- An accessor method is trivial if it has no parameters and contains one `return` statement that returns a non-static data member or a reference to a non-static data member.

  The return type of the method must exactly match or be a reference to the type of the data member.
- A mutator method is trivial if it has a `void` return type, one parameter, and contains one assignment statement that assigns the parameter to a non-static data member.

  The parameter type must exactly match or be a reference to the type of the data member.

The checker flags trivial accessor and mutator methods defined outside their classes without the `inline` keyword.

The checker does not flag template methods or virtual methods.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Trivial Accessor and Mutator Methods Defined Outside Classes

```
class PhysicalConstants {
    public:
       double getSpeedOfLight() const;
       void setSpeedOfLightInMedium(double newSpeed);
```

```
        double getRefractiveIndexGlass() { //Compliant
            return refractiveIndexGlass;
        }
    private:
        double speedOfLight;
        double refractiveIndexGlass;
};

double PhysicalConstants::getSpeedOfLight() const{ //Noncompliant
    return speedOfLight;
}

void PhysicalConstants::setSpeedOfLightInMedium(double newSpeed) {//Noncompliant
    speedOfLight = newSpeed;
}
```

In this example, the accessor methods `getSpeedOfLight` and `getRefractiveIndexGlass` are trivial. The `getSpeedOfLight` method is defined outside its class and is noncompliant. The `getRefractiveIndexGlass` method is defined inside the class definition and complies with the rule.

The trivial mutator method `setSpeedOfLightInMedium` is also defined outside the class definition and violates the rule.

### Check Information

**Group:** Basic concepts
**Category:** Advisory, Automated

## Version History

**Introduced in R2020b**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A3-3-1

Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file

## Description

### Rule Definition

*Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file.*

### Rationale

If you declare a function or object in a header file, it is clear that the function or object is meant to be accessed in multiple translation units. If you intend to access the function or object from a single translation unit, declare it `static` or in an unnamed namespace.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Declaration in Header File Missing

This example uses two files:

- `decls.h`:

  ```
  extern int x;
  ```
- `file.cpp`:

  ```
  #include "decls.h"

  int x = 0;
  int y = 0; //Noncompliant
  static int z = 0;
  ```

In this example, the variable `x` is declared in a header file but the variable `y` is not. The variable `z` is also not declared in a header file but it is declared with the `static` specifier and does not have external linkage.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A3-3-2

Static and thread-local objects shall be constant-initialized

## Description

### Rule Definition

*Static and thread-local objects shall be constant-initialized.*

### Rationale

Static and thread-local objects are initialized at the start of code execution. The C++ language standard only partially defines the initialization order of multiple static or thread-local objects and the order can change from build to build. If you initialize a static or thread-local object from another such object, the compiler might access the latter object before it is initialized. To avoid access before initialization, initialize static and thread-local objects by using objects that evaluate to a constant at compile time. Initialization with constants occurs before initialization with variables and often happens at compile time.

This rule applies to global variables, static variables, static class member variables, and static function-scope variables.

### Polyspace Implementation

Polyspace flags initializations of static or thread-local objects using initializers and constructors that do not evaluate to constants at compile time. To constant-initialize static or thread-local objects, use:

- A `constexpr` constructor with only constant arguments
- A constant expression
- A value

Because string objects use dynamic memory allocation of unknown size, the compiler cannot evaluate them at compile time. Polyspace flags initialization of string objects irrespective of whether you specify an initializer.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Initializing Static and Thread-Local Objects

```
#include <cstdint>
#include <limits>
#include <string>
class A{
    //..
public:
    constexpr A(){
        //...
    }
};
class B{
    //..
public:
    B(){
        //...
    }
};
const int global_const_a = 10;              // Compliant
const int global_const_b = global_const_a;  // Compliant
int global_a = 10;                          // Compliant
int global_b = global_a;                    // Noncompliant
static std::string global_name = "Name";    // Noncompliant
static std::string global_id;               // Noncompliant
char *ptr = "hello world";                  // Compliant
char arr_up[3] = {'U','p','\0'};            // Compliant
char container[10];                         // Compliant
extern const int global_extern_c;
const int global_const_c = global_extern_c; // Noncompliant
static A obj1{};                            //Compliant
static B obj2{};                             //Noncompliant
main()
{

    //
}
```

Polyspace flags the initialization of:

- `global_b` by `global_a` because whether `global_b` evaluates to a constant at compile time depends on the order in which these variables are initialized.
- `global_name` and `global_id` because the compiler cannot evaluate constructor for string objects at compile time.
- `global_const_c` by the `extern` variable `global extern_c` because the compiler cannot evaluate `extern` variables at compile time.
- `obj2`, which calls the constructor `B::B()`, because the constructor is not specified as `constexpr`.

Polyspace does not flag the initialization of:

- `global_const_b` by `global_const_a` because the compiler can evaluate these objects at compile time regardless of their initialization order.
- `global_const_a` and `global_a` by literals because the compiler can evaluate literals at compile time.
- Global character pointers and arrays by literal initializers because the compiler can allocate static memory at compile time.
- `obj1`, which calls the constructor `A::A()`, because the constructor is specified as `constexpr`.

## Check Information

**Group:** Basic concepts
**Category:** Required, Automated

## Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A3-8-1

An object shall not be accessed outside of its lifetime

## Description

### Rule Definition

*An object shall not be accessed outside of its lifetime.*

### Rationale

The lifetime of an object begins when it is created by its constructor. The lifetime ends when the object is deleted. Accessing a variable before its construction or after its destruction can lead to undefined behavior. Depending on the context, many operations might inadvertently access an object outside its lifetime. Examples of such operations include:

- Noninitialized pointer: You might inadvertently access a pointer before assigning an address to it. This operation accesses an object before its lifetime and results in accessing an unpredictable memory location. The best practice is to initiate a pointer by using `nullptr` during its declaration.
- Noninitialized variable: You might inadvertently read a variable before it is initialized. This operation accesses an object before its lifetime and results in reading a garbage value that is unpredictable and useless. The best practice is to initiate a variable during its declaration.
- Use of previously deallocated pointer: You might access the dynamically allocated memory of a pointer after deallocating the memory. Trying to access this block of memory accesses an object after its lifetime and results in unpredictable behavior or even a segmentation fault. To address this issue, set the deallocated pointer to `nullptr`, and then to check if a pointer is `nullptr` before accessing it. Alternatively, use a `std::unique_ptr` instead of a raw pointer. Because you do not need to deallocate the allocated memory for a `std::unique_ptr` explicitly, you can avoid inadvertently accessing the deallocated memory.
- Pointer or reference to stack variable leaving scope: You might assign a nonlocal pointer to a local object. For instance:

  - A nonlocal or global pointer is assigned to a variable that is local to a function.
  - A passed-by-reference function parameter, such as a pointer, is assigned to a variable that is local to a function.
  - A pointer data member of a class is assigned to a variable that is local to a function.

  Once the local variable goes out of scope, their corresponding memory blocks might hold garbage or unpredictable values. Accessing pointers to these memory locations accesses an object after its lifetime and might result in undefined or unpredictable behavior. The best practice is to not assign nonlocal pointers to local objects.
- Modifying object with temporary lifetime: You might attempt to modify a temporary object returned by a function call. Modifying temporary objects is an undefined behavior that might lead to abnormal program termination depending on the hardware and software that you use. The best practice is to assign the temporary objects in local variables, and then modifying the local variables.

Avoid operations that might access an object outside of its lifetime.

#### Polyspace Implementation

Polyspace checks for these scenarios where an object might be accessed outside of its lifetime:

- Noninitialized pointer: Polyspace flags a pointer if it is not assigned an address before it is accessed.
- Noninitialized variable: Polyspace flags a variable if it is not initialized before its value is read.
- Use of previously deallocated pointer: Polyspace flags an operation where you access a block of memory after deallocating the block, for instance, by using the `free()` function or the `delete` operator.
- Pointer or reference to stack variable leaving scope: Polyspace flags a local variable when a pointer or reference to it leaves its scope. For example, a local variable is flagged when:

  - A function returns a pointer to the local variable
  - A global pointer is pointed to the local variable
  - A pass-by-reference function parameter, such as a pointer, is pointed to the local variable
  - A pointer data member of a class is pointed to the local variable

  Polyspace assumes that the local objects within a function definition are in the same scope.
- Accessing object with temporary lifetime: Polyspace flags an operation where you access a temporary object that is returned by a function call.

#### Extend Checker

You can extend the checker in the following ways:

- Polyspace does not flag passing pointers to noninitialized variables to functions. To detect noninitialized variables that are passed to functions by pointers, extend the checker by using the option `-code-behavior-specification`. See “Extend Checkers for Initialization to Check Function Arguments Passed by Pointers”.
- If a variable in your code is non-initialized only for certain system input values, you can see one possible combination of input values causing the defect. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

#### Avoid Accessing Noninitialized Pointer

This example shows how Polyspace flags accessing pointers that are not assigned to an address.

```
#include <cstdlib>

int* Noncompliant(int* prev)
{
    int j = 42;
    int* pi;
    if (prev == nullptr){
        pi = new int;
        if (pi == nullptr)
        return nullptr;
    }
    *pi = j; //Noncompliant
    return pi;
}
int* Compliant(int* prev)
{
    int j = 42;
    int* pi;
    if (prev == nullptr){
        pi = new int;
        if (pi == nullptr)
        return nullptr;
    }
    else
    pi = prev;
    *pi = j;//Compliant
    return pi;
}
int* AltCompliant(int* prev)
{
    int j = 42;
    int* pi=nullptr;
    if (prev == nullptr){
        pi = new int;
        if (pi == nullptr)
        return nullptr;
    }
    else
    if(pi!= nullptr)
    *pi = j;//Compliant
    return pi;
}
```

Polyspace flags the pointer pi in Noncompliant() because pi is accessed before an address is assigned to it when prev is not NULL. You can address this issue in various ways. For instance:

- Initiate pi before the statement *pi = j. Assignment to pi in Compliant() is not flagged because pi is initiated by prev before it is accessed.
- Initiate pi by using nullptr during its declaration. Assignment to pi in AltCompliant() is not flagged because pi is initiated by nullptr during its declaration.

### Avoid Accessing Noninitialized Variable

This example shows how Polyspace flags accessing noninitialized variables.

```
int Noncompliant(void)
{
    extern int getsensor(void);
    int command;
    int val;
    command = getsensor();
    if (command == 2){
        val = getsensor();
    }
    return val;//Noncompliant

}
int Compliant(void)
{
    extern int getsensor(void);
    int command;
    int val=0;//Initialization
    command = getsensor();
    if (command == 2){
        val = getsensor();
    }
    return val;//Compliant
}
```

Polyspace flags the statement `return val` in `Noncompliant()` because this statement accesses `val` before the variable is initialized when `command` is not equal to `2`. You can address this issue in several ways. For instance, initialize the variable `val` to zero during its declaration, as shown in `Compliant()`. By initializing the variable during declaration, it is initialized in all execution paths, making the statement `return val` compliant with this rule.

### Avoid Using Previously Deallocated Pointer

This example shows how Polyspace flags accessing pointers that might point to already released memory blocks.

```
#include <memory>
int Noncompliant(double base_val, double shift){
    double j;
    double* pi = new double;
    if (pi == nullptr)
    return 0;
    *pi = base_val;
    //...
    delete pi;
    //...
    j = *pi + shift;//Noncompliant
    return j;
}
int Compliant(double base_val, double shift){
    double j;
    std::unique_ptr<double>   pi(new double(3.1416));
    if (pi == nullptr)
    return 0;
    *pi = base_val;
    j = *pi + shift;
    return j;
}
```

In the function `Noncompliant()`, a pointer `pi` is declared and initialized by using the operator `new`. Later, the dynamically allocated memory is deallocated by using the operator `delete`. The deallocated pointer is then inadvertently accessed in the statement `j = *pi + shift;`. Polyspace flags this statement. You can address this issue in various ways. For instance, you might want to deallocate the allocated resource after performing all relevant operations. Alternatively, you can use smart pointers instead of raw pointers. In `Compliant()`, the pointer `pi` is declared as a `std::unique_ptr`. The acquired resources for `pi` are automatically deallocated at the end of

Compliant() by calling its destructor. Because the memory allocated for pi is not accessed after it is deallocated, Compliant() is compliant with this rule.

### Avoid Emitting Pointers or References to Local Variables to Outer Scopes

This example shows how Polyspace flags operations where pointers to local variables might escape to outer scopes.

```
int* Noncompliant1(void) {
    int ret = 0; //Noncompliant
    return &ret ;
}
auto Noncompliant2(int var) {
    int rhs = var; //Noncompliant
    auto adder = [&] (int lhs) {
        return (rhs + lhs);
    };
    return adder;
}
int Compliant1(void) {
    int ret = 0; //Compliant
    return ret ;
}
auto Compliant2(int var) {
    int rhs = var; //Compliant
    auto adder = [=] (int lhs) {
        return (rhs + lhs);
    };
    return adder;
}
```

- The function Noncompliant1() returns a pointer to the local variable ret. The local variable ret is deleted as soon as Noncompliant() finishes execution. The returned pointer points to an unpredictable value. Such operations are noncompliant with the rule. You can fix this issue by returning local variables by value, as shown in Compliant().
- The function Noncompliant2() returns a lambda expression, which captures the local variable rhs by reference. This reference dereferences to an unpredictable value because rhs is deleted when the function Noncompliant2() finishes execution. You can fix this issue by capturing local variables by copy in the lambda expression, as shown in Compliant2().

### Avoid Accessing Temporary Objects

This example shows how Polyspace flags operations that might access temporary objects that are created by a function call.

```
#include<vector>
struct S_Array{
    int t;
    int a[5];
};
struct S_Array Factory(void);
std::vector<int> VectorFactory(int aNumber);
int Noncompliant(void) {

    return ++(Factory().a[0]); //Noncompliant
}
int Compliant(void) {
    auto tmp = Factory();
    return ++(tmp.a[0]); //Compliant
}
int Compliant2(void) {
    return ++(VectorFactory(5)[1]); //Compliant
}
```

In `Noncompliant()`, the call to `Factory()` creates a temporary object. Modifying this object is noncompliant with this rule. Polyspace flags the statement `return ++(Factory().a[0])`. You can address this issue in various ways. For instance, you can assign the temporary object to a local variable before modifying it, as shown in `Compliant()`. Alternatively, use smart containers such as `std::vector` as shown in `Compliant2()`. Containers such as `std::vector` manage their own lifetime and have move semantics. Polyspace does not flag the statement `return ++(VectorFactory(5)[1]);`.

## Check Information

**Group:** Basic concepts
**Category:** Required, Non-automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A3-9-1

Fixed width integer types from <cstdint>, indicating the size and signedness, shall be used in place of the basic numerical types

## Description

### Rule Definition

*Fixed width integer types from <cstdint>, indicating the size and signedness, shall be used in place of the basic numerical types.*

### Polyspace Implementation

Only allows use of basic types through direct `typedefs`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A4-5-1

Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=

## Description

### Rule Definition

*Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.*

### Rationale

In C++, enumerations such as `enum` or `enum class` have implementations defined behavior. For instance, their underlying type can be any integral type, including `short` or `char`. If you use enumerations as operands to arithmetic operators such as + or `-`, they are converted to their underlying type. Because the underlying type of an enumeration is implementation dependent, outcome of arithmetic operations using enumerations as operands is unpredictable. To avoid unpredictable and non-portable code, use enumerations as operands to only these operators:

- Subscript operator `[]`
- Assignment operator =
- Equality operators == and `!=`
- The Unary & operator
- The relational operators `<, <=,>,>=`

You can use enumerations as operands to the built in or overloaded instances of only the above operators. Note that Bitmask type enumerations are an exception to this rule. That is, you can use Bitmask type enumerations as operands to any operators.

### Polyspace Implementation

Enumerations are valid operands to only the operators listed above. Polyspace flags enumerations when they are used as operands to any other operators. Note that Polyspace makes no exception for BitmaskType enumerations.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Do Not Use Enumerations as Operands to Arithmetic Operators

```
#include <cstdint>
enum Color : std::uint8_t {  Red,  Green,  Blue,  ColorsCount};
enum class Car : std::uint8_t { Model1, Model2, Model3, ModelsCount};
enum BMT {Exec = 0x1,Write  = 0x2,Read = 0x4};
Car operator+(Car lhs, Car rhs)
{
    return Car::Model3;
}
Color operator|=(Color lhs, Color rhs)
{
    return rhs;
};
void F1()
{
    Car car = Car::Model1;
    Color color = Red;
    if (color == Green) {                  // Compliant
    }

    if (color == (Red + Blue)) {           // Noncompliant
    }

    if (color < ColorsCount) {             //Compliant
    }
    if (car == (Car::Model1 + Car::Model2)) // Noncompliant
    {
    }
    Color value;
    value = (Color)(Red | 3);              // Noncompliant
    value |= Blue;                         // Noncompliant
    value = (Color)0;                      // Compliant
    if (value & Blue) {};                  // Noncompliant
    value = (Color)(Blue * value       );  // Noncompliant
    value = (Color)(Red << 3);             // Noncompliant
    value = (Color)(Red >> 12);            // Noncompliant
    BMT bitmask1 = (BMT)(Exec + Write);    // Noncompliant
    BMT bitmask2 = (BMT)(Exec | Write);    // Noncompliant
}
```

The line `BMT bitmask1 = (BMT)(Exec + Write);` adds two enumerators and assigns the result to the `enum` object `bitmask1`. The addition operation implicitly converts the enumerators into their underlying type. Because the underlying type of enumerators are implementation dependent, the outcome of this code can be unpredictable. Polyspace flags the enumerators that are operands to the built in + operator.

Polyspace treats both built in and overloaded operators similarly. For example, Polyspace flags the operands in the operation `Car::Model1 + Car::Model2`, even though the + operator is overloaded for the `enum class Car`.

### Check Information

**Group:** Standard conversions
**Category:** Required, Automated

## Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A4-7-1

An integer expression shall not lead to data loss

## Description

### Rule Definition

*An integer expression shall not lead to data loss.*

### Rationale

A data loss might occur if you perform an explicit cast or if your integer expression results in an implicit conversion, an overflow, an underflow, or a wraparound. For instance:

- An implicit conversion from `uint16_t` to `uint8_t` discards the high byte of the larger data type.
- An arithmetic expression with signed integers that results in an overflow is undefined behavior.

To make sure that no unexpected data loss occurs:

- Avoid integral type conversions by performing all operations in a uniform type.
- Use appropriate guards (such as asserts and if statements) to handle other possible causes of data loss.

### Polyspace Implementation

- Polyspace flags these integral expressions that might result in data loss:
  - Operation on a signed or an unsigned integer variables that results in an overflow.
  - Assignment of a compile-time constant to signed or unsigned integer variables whose data type cannot accommodate the value of that constant.
  - Conversion of a signed (unsigned) integer to a narrower signed (unsigned) integer type.
  - Conversion of an unsigned integer to a signed integer.
  - Shift operation that results in a value that cannot be represented by the result data type.
- Polyspace does not flag the use of `static_cast` to cast to a narrower type. The software assumes that these conversion are intentional even if they might result in data loss.

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Integer Expressions that Lead to Data Loss

```
#include <iostream>
#include <cstdint>
#include <stdexcept>
#include <climits>

uint8_t sum(uint8_t a, uint8_t b) noexcept
{
    return (a + b); //Non-compliant
}

uint8_t sum_check(uint8_t a, uint8_t b)
{
    if (b > UCHAR_MAX - a) {
        throw std::range_error("Operation overflows");
    }
    return (a + b); // Compliant
}

int16_t increment(int16_t var)
{
    return ++var; //Non-compliant
}


void func()
{

    uint8_t small_sum = sum(50, 50);
    uint8_t large_sum = sum(150, 150);
    try {
        uint8_t large_sum_check = sum_check(150, 150);
    } catch (std::range_error&) {
        //Handle error
    }

    int16_t max_var = increment(SHRT_MAX);

}
```

In this example, Polyspace flags:

- The return statement of `sum()` because the second call to `sum()` to initialize `large_sum` results in an overflow. The sum of the input parameters exceeds the size of the return type (`uint8_t`).
- The integer expression of `increment()` because the call when initializing `max_var` attempts to increment `SHRT_MAX`.

Polyspace does not flag the return statement of `sum_check` because the function checks the range of its inputs and throws an error when `large_sum_check` is initialized.

## Check Information

**Group:** Standard conversions

**Category:** Required, Automated

## Tips

Polyspace Bug Finder makes certain assumptions about the values of inputs. See “Bug Finder Analysis Assumptions”.

At the cost of a possibly longer runtime, you can perform a more exhaustive analysis where all values of function inputs are considered when showing defects, including inputs of uncalled functions. See `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`.

# Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A4-10-1

Only nullptr literal shall be used as the null-pointer-constraint

## Description

### Rule Definition

*Only nullptr literal shall be used as the null-pointer-constraint.*

### Rationale

`nullptr` was introduced in C++11 to support the concept of a pointer that does not point to a valid object. Before C++11, the macro `NULL` and the constant 0 were the only ways to define the null pointer constant. Using `nullptr` to indicate null-pointers has several advantages over using `NULL` or `0`. For instance:

- `nullptr` can be used with any type of null-pointer without requiring an implicit cast.
- `nullptr` literals allow parameter forwarding by using a template function.

`NULL` is a macro that expands to an integer `0` which is cast into `void*` type. Using `NULL` or `0` to indicate null-pointers is contrary to developer expectation. If code expecting `nullptr` encounters `NULL` or `0` instead, it might lead to confusion or unexpected behavior.

### Polyspace Implementation

Polyspace flags the use of `NULL` or `0` instead of `nullptr` to indicate a null-pointer. This rule does not check for conversion between `NULL` and `0`. See `AUTOSAR C++14 Rule M4-10-1`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `NULL` or `0` as Alternatives to `nullptr`

```
#include <cstdint>
#include <cstddef>


void foo(int*);
void foo2(int*);

void bar() {
    foo(NULL);    //Noncompliant
    foo2(0);      //Noncompliant
    foo(nullptr); //Compliant
}
```

In this example, the rule is violated when the macro NULL or the constant 0 is used as a null-pointer instead of nullptr.

## Check Information

**Group:** Standard conversions
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

Check AUTOSAR C++ 14 (-autosar-cpp14) | AUTOSAR C++14 Rule M4-10-1 | AUTOSAR C++14 Rule M4-10-2

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-0-1

The value of an expression shall be the same under any order of evaluation that the standard permits

## Description

### Rule Definition

*The value of an expression shall be the same under any order of evaluation that the standard permits.*

### Rationale

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

### Polyspace Implementation

Polyspace raises a violation if an expression satisfies any of these conditions:

- The same variable is modified more than once in the expression or it is both read and written.
- The expression allows more than one order of evaluation.
- The expression contains a single `volatile` object that occurs multiple times.
- The expression contains more than one `volatile` object.

Because `volatile` objects can change their value at anytime, an expression containing multiple `volatile` variables or multiple instances of the same `volatile` variable might have different results depending on the order of evaluation.

When checking an expression containing function calls, Polyspace checks the body of the functions. If the function body contains calls to other functions, their bodies are not checked. For instance:

```
int f1(){return 1;}
int f2(){return g();}
int g(){return 2;}

void foo(void){
    int z = f1()+f2();
}
```

Here, when checking the expression `z = f1()+f2()`, Polyspace checks the body of `f1()` and `f2()`. The body of `g()` is not checked.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])
```

```
void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         // Compliant
    COPY_ELEMENT (i++);    // // Non-compliant
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

### Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
    f ( i++, i );                 // Noncompliant
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

### Multiple `volatile` Objects in an Expression

```
volatile int a, b;
int mathOp(int x, int y);

int foo(void){
    int temp = mathOp(5,a) + mathOp(6,b);//Noncompliant
    return temp * mathOp(a,a);//Noncompliant
}
```

In this example, this rule is violated twice.

- The declaration of `temp` uses two `volatile` objects in the expression. Because the value of `volatile` objects might change at any time, the expression might evaluate to different values depending on the order of evaluation. Polyspace flags the second `volatile` object in the expression.
- The `return` statement uses the same `volatile` object twice. Because the expression might have different results depending on the order of evaluation, Polyspace raises this defect.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-0-2

The condition of an if-statement and the condition of an iteration statement shall have type bool

## Description

### Rule Definition

*The condition of an if-statement and the condition of an iteration statement shall have type bool.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A5-0-3

The declaration of objects shall contain no more than two levels of pointer indirection

## Description

### Rule Definition

*The declaration of objects shall contain no more than two levels of pointer indirection.*

### Rationale

If you use pointers with more than two levels of indirection, a developer reading the code might find it difficult to understand the behavior of the code.

### Polyspace Implementation

Polyspace flags all declarations of objects that contain more than two levels of pointer indirection.

- If you use type aliases, the checker includes pointer indirections from the alias in the evaluation of the level of indirection. For instance, in this code snippet, the declaration of `var` is non-compliant. The type of `var` is `const` pointer to a `const` pointer to a pointer to `char`, which is three levels of pointer indirection. The declaration of `var2` has two levels of pointer indirection and is compliant.

  ```
  using ptrToChar = char*;

  void func()
  {
      ptrToChar* const* const var = nullptr; //Non-compliant, 3 levels of indirection
      char* const* const var2 = nullptr; //Compliant, 2 levels of indirection
      //...
  }
  ```
- If you pass an array to a function, the conversion of the array to a pointer to the first element of the array is included in the evaluation of the level of indirection. For instance, in this code snippet, parameter `arrParam` is non-compliant. The type of `arrParam` is a pointer to a pointer to a pointer to `char` (three levels of pointer indirection). The declaration of `arrVar` is compliant because `arrVar` has type array of pointer to pointer to char (two levels of pointer indirection).

  ```
  void func(char** arrParam[])  //Non-compliant
  {
      //...
      char** arrVar[5]; //Compliant
  }
  ```

This checker does not flag the use of objects with more than two levels of indirection. For instance, in this code snippet, the declaration of `var` is non-compliant, but the evaluation of the size of `var` is compliant.

```
#include<iostream>


using charToPtr = char*;

void func()
{
    charToPtr* const* const var = nullptr; //Non-compliant
```

```
    std::cout << sizeof(var) << std::endl; //Compliant

}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A5-0-4

Pointer arithmetic shall not be used with pointers to non-final classes

## Description

### Rule Definition

*Pointer arithmetic shall not be used with pointers to non-final classes.*

### Polyspace Implementation

Reports pointer arithmetic and array like access on expressions whose pointed type is used as a base class.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A5-1-1

Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead

## Description

### Rule Definition

*Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.*

### Rationale

Improve the readability and maintainability of code by using symbolic names. Literal constants do not clearly indicate what the constant represents

### Polyspace Implementation

Polyspace flags use of literal values other than those with the data type `char` in expressions and `case` clauses of a `switch` statement.

Polyspace does not flag the use of literal values in logging mechanisms.

Polyspace does not flag the use of literal values '`0`' and '`1`' in expressions, as they are often part of the logic of the code. For instance, '`0`' represents a `NULL` pointer.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Using Literal Values

```
double foo(void){
    constexpr int speedLimit = 65;
    constexpr double coeff = 0.2;
    int flag{0};
    int negOne{-1};
    //...
    return (flag)?speedLimit*coeff*negOne + 35 //Noncompliant
            : speedLimit*coeff*negOne - 35; //Noncompliant
}

double bar(void){
    constexpr int speedLimit = 65;
    constexpr double coeff = 0.2;
    int flag{0};
    int negOne{-1};
    int addedValue = 65;
    return (flag)?speedLimit*coeff*negOne + addedValue //Compliant
```

```
            : speedLimit*coeff*negOne - addedValue; //Compliant
}
```

In this example, Polyspace flags the use of literal values in the `return` expression of `foo`. In `bar`, the return expression uses symbolic names, which makes code more readable. The use of literals to initiate objects does not violate this rule.

## Check Information

**Group:** Expressions
**Category:** Required, Partially automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-1-2

Variables shall not be implicitly captured in a lambda expression

## Description

### Rule Definition

*Variables shall not be implicitly captured in a lambda expression.*

### Rationale

In a lambda expression, you have the option to capture variables implicitly. For instance, this lambda expression

```
[&](std::int32_t var) {
   sum+ = var;
}
```

indicates that all local variables in the calling context are captured by reference. However, it is not immediately clear from this lambda expression:

- If a variable in the body of the expression comes from the calling context.

  For instance, in the preceding lambda expression, it is not clear if `sum` is captured from the calling context or is a global variable.
- If all variables captured from the calling context are used and whether the variables are modified or just read (If the variables are read, a by-copy capture is preferred).

If you capture variables explicitly in a lambda expression, you have more control on whether to capture by reference or copy. In addition, you or a reviewer can read the lambda expression and determine whether a variable was captured from the calling context.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Lambda Expressions with Implicit and Explicit Capture

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <cstdint>

void addEvenNumbers(std::vector<std::int32_t> numbers)
{
    std::int64_t sum = 0;
    std::int32_t divisor = 2;
    for_each(numbers.begin(), numbers.end(), [&] (std::int32_t y) //Noncompliant
    {
        if (y % divisor == 0)
        {
            std::cout << y << std::endl;
            sum += y;
        }
    });

    std::cout << sum << std::endl;
}

void addOddNumbers(std::vector<std::int32_t> numbers)
{
    std::int64_t sum = 0;
    std::int32_t divisor = 2;
    for_each(numbers.begin(), numbers.end(), [&sum, divisor] (std::int32_t y) //Compliant
    {
        if (y % divisor != 0)
        {
            std::cout << y << std::endl;
            sum += y;
        }
    });

    std::cout << sum << std::endl;
}
```

The lambda expression in the `addEvenNumbers` function captures all local variables in the calling context implicitly by reference and violates this rule. Some of the issues are:

- Unless you go through the body of the expression, it is not clear which variables are used.
- Though the variable `divisor` is only read and not modified, it is captured by reference. A by-copy capture is preferred.

The lambda expression in the `addOddNumbers` function captures each variable explicitly and does not violate this rule. Without looking at the body of the lambda expression, you can determine which variables are intended to be modified in the expression.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A5-1-3

Parameter list (possibly empty) shall be included in every lambda expression

## Description

### Rule Definition

*Parameter list (possibly empty) shall be included in every lambda expression.*

### Rationale

You do not have to include a parameter list in a lambda expression. For instance, this expression is syntactically valid and indicates a closure that can be called without parameters:

```
[&counter] {
   ++counter;
}
```

However, without the `()`, you or a reviewer might not recognize this as a function object. It is visually clearer to use the parameter list `(...)` even when the list is empty. For instance:

```
[&counter]() {
   ++counter;
}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Lambda Expressions Without Parameter List

```
#include <cstdint>

void func() {
    std::int32_t count = 0;

    auto lambda1 = [&count] {++count;}; //Noncompliant
    auto lambda2 = [&count] () { //Compliant
        ++count;
    };
}
```

The lambda expression assigned to `lambda1` does not have a parameter list and violates the rule. The issue is fixed when the same lambda expression is assigned to `lambda2`.

## Check Information

**Group:** Expressions

**Category:** Required, Automated

## Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-1-4

A lambda expression object shall not outlive any of its reference-captured objects

## Description

### Rule Definition

*A lambda expression object shall not outlive any of its reference-captured objects.*

### Rationale

The rule flags cases where a lambda expression captures an object *by reference* and you can potentially access the captured object outside its scope. This situation happens if the lambda expression object outlives the object captured by reference.

For instance, consider this function `createFunction`:

```
std::function<std::int32_t()> createFunction() {
   std::int32_t localVar = 0;
   return ([&localVar]() -> std::int32_t {
       localVar = 1;
       return localVar;
   });
}
```

`createFunction` returns a lambda expression object that captures the local variable `localVar` *by reference*. The scope of `localVar` is limited to `createFunction` but the lambda expression object returned has a much larger scope.

This situation can result in an attempt to access the local object `localVar` outside its scope. For instance, when you call `createFunction` and assign the returned lambda expression object to another object `aFunction`:

```
auto aFunction = createFunction();
```

and then invoke the new object `aFunction`:

```
std::int32_t someValue = aFunction();
```

the captured variable `localVar` is no longer in scope. Therefore, the value returned from `aFunction` is undefined.

If a function returns a lambda expression, to avoid accessing a captured object outside its scope, make sure that the lambda expression captures all objects by copy. For instance, you can rewrite `createFunction` as:

```
std::function<std::int32_t()> createFunction() {
   std::int32_t localVar = 0;
   return ([localVar]() mutable -> std::int32_t {
       localVar = 1;
       return localVar;
   });
}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Lambda Expressions that Accesses variables Out-of-Scope

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd; //Noncompliant
  auto adder = [&] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void foo() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}

auto createMultiplier(int amountToMultiply) {
  int multiplyThis = amountToMultiply; //Compliant
  auto adder = [=] (int initialAmount) {
      return (initialAmount + multiplyThis);
  };
  return adder;
}


void bar() {
  auto MultiplyByTwo = createMultiplier(2);
  int res = MultiplyByTwo(10);
}
```

In this example, the `createAdder` function defines a lambda expression `adder` that captures the local variable `addThis` by reference. The scope of `addThis` is limited to the `createAdder` function. When the object `AddByTwo`, which is returned by `createAdder`, is called, a reference to the variable `addThis` is accessed outside its scope. When accessed in this way, the value of `addThis` is undefined.

The issue does not occur with the `createMultiplier` function, which returns a lambda expression that captures local variables by copy.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-1-6

Return type of a non-void return type lambda expression should be explicitly specified

## Description

### Rule Definition

*Return type of a non-void return type lambda expression should be explicitly specified.*

### Rationale

A compiler can deduce the return type of a lambda expression based on the type of the return expression. For instance, if a lambda expression does not return anything, the compiler deduces that the return type is `void`.

Specifying a return type when you declare a lambda expression is optional. For non-void return type lambda expressions, if you do not specify a return type explicitly, a developer reading your code might be confused about which type the lambda expression returns.

An explicit return type also reinforces type checking when the compiler generates an implicit conversion from the type of the returned expression to the expected return type.

### Polyspace Implementation

Polyspace flags lambda expressions with non-void-return types if the return type is not specified explicitly.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Return Type not Specified for Lambda Expression with Non-Void Return Type

```
#include<iostream>
#include <cstdint>
#include <cstdio>

void  func()
{
    std::uint8_t TARGET = 10;
    auto lambda_incr = [&](std::uint8_t x) -> std::uint8_t {//Compliant
        while (x < TARGET)
            x++;
        return x;

    };
    auto lambda_decr = [&](std::uint8_t y) { //Non-compliant, returned type is not specified
        while (y > TARGET)
            y--;
        return y;

    };
    char exp[] = "hello.";
    auto lambda3 = [exp]() { //Compliant, void return type.
        std::cout << exp << std::endl;
    };
    auto x = lambda_incr(5);
    auto y = lambda_decr(11);
    lambda3();


}
```

In this example, Polyspace flags lambda expression `lambda_decr` because no return type is specified. Polyspace does not flag `lambda3` even though no return type is specified because the expression does not return anything (void return type).

## Check Information

**Group:** Expressions
**Category:** Advisory, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-1-7

A lambda shall not be an operand to decltype or typeid

## Description

### Rule Definition

*A lambda shall not be an operand to decltype or typeid.*

### Rationale

According to the C++ Standard, the type of a lambda expression is a unique, unnamed class type. Because the type is unique, another variable or expression cannot have the same type. Use of `decltype` or `typeid` on a lambda expression indicates that you expect a second variable or expression to have the same type as the operand lambda expression.

Both `decltype` and `typeid` return the data type of their operands. Typically the operators are used to:

- Assign a type to another variable. For instance:

  ```
  decltype(var1) var2;
  ```

  creates a variable `var2` with the same type as `var1`.
- Compare the types of two variables. For instance:

  ```
  (typeid(var1) == typeid(var2))
  ```

  compares the types of `var1` and `var2`.

These uses do not apply to a lambda expression, which has a unique type.

### Polyspace Implementation

The rule checker flags uses of `decltype` and `typeid` with lambda expressions.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `typeid` on Lambda Expressions

```
#include <cstdint>
#include <typeinfo>

 void func()
 {
 auto lambdaFirst = []() -> std::int8_t { return 1; };
```

```
 auto lambdaSecond = []() -> std::int8_t { return 1; };

 if (typeid(lambdaFirst) == typeid(lambdaSecond))
     {
     // ...
     }
 }
```

The use of `typeid` on lambda expressions can lead to unexpected results. The comparison above is false even though `lambdaFirst` and `lambdaSecond` appear to have the same body.

#### Correction – Assign Lambda Expression to Function Object Before Using typeid

One possible correction is to assign the lambda expression to a function object and then use the `typeid` operator on the function objects for comparison.

```
#include <cstdint>
#include <functional>
#include <typeinfo>

 void func()
 {
 std::function<std::int8_t()> functionFirst = []() { return 1; };
 std::function<std::int8_t()> functionSecond = []() { return 1; };

 if (typeid(functionFirst) == typeid(functionSecond))
     {
     // ...
     }
 }
```

## Check Information

**Group:** Lexical Conventions
**Category:** Required, Automated

# Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-1-8

Lambda expressions should not be defined inside another lambda expression

## Description

### Rule Definition

*Lambda expressions should not be defined inside another lambda expression.*

### Rationale

Developers can use lambda expressions to write anonymous function objects that contain a few lines of code. Nesting lambda expression reduces the readability of the code because the body of a lambda expression is typically in the line where it is used. For instance, the `find_if` algorithm takes a unary predicate as one of its arguments. A developer can use a lambda expression to define a predicate condition in the declaration of `find_if`. In this code snippet, the `find_if` algorithm returns the first member of a vector of integers that is greater than 2 and that is even.

```
std::vector<int> v = { 1, 2, 3, 4 };
std::find(v.begin(), v.end(),
    [](int val) { return val>2 && val%2==0;});
```

### Polyspace Implementation

Polyspace flags lambda expressions that are defined inside another lambda expression. Polyspace also highlights the closest nesting lambda expression.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Nested Lambda Expressions

```
#include<iostream>
#include<vector>
#include<algorithm>

int main()
{

    using namespace std;

    vector<int> v {1, 2, 3, 4};


    vector<int>::iterator it = v.begin();
    while (it != v.end()) {
        auto evenGreater2 = [](int val) {

            return [](int val2) { //Noncompliant
                return val2 % 2 == 0;
            }(val)&& (val) > 2;

        }(*it);
        if (evenGreater2) {
            cout << *it << endl;
            break;
        }
        ++it;
    }

}
```

In this example, Polyspace flags the lambda expression that checks whether a value is even (`[](int val2) { return val2 % 2 ==0; }`) because it is nested inside another lambda expression that also checks whether a value is greater than 2.

## Check Information

**Group:** Expressions
**Category:** Advisory, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-1-9

Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression

## Description

### Rule Definition

*Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression.*

### Rationale

When you reuse an unnamed lambda expression, you insert the body of that lambda expression wherever you invoke it in your code. This code duplication might result in maintainability issues when you make changes, as you might misidentify which lambda expressions are identical when applying those changes. The code duplication also decreases the readability of your code.

### Polyspace Implementation

After the first use of an unnamed lambda expression, Polyspace flags each subsequent uses of an identical lambda expression. For instance, if you reuse the same lambda expression three times, Polyspace flags the second and third uses of the lambda expression as separate violations. Polyspace also highlights the first use of the unnamed lambda expression in your source code.

Polyspace does not flag the reuse of global scope lambda expressions.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Reuse of Unnamed Lambda Expression

```
#include<vector>
#include<algorithm>


void func1(std::vector<int>& v)
{
    if (none_of(v.begin(), v.end(),
    [](int i) {return i % 2 == 1;})) {
        //Handle error
    }

    int odds = std::count_if(v.begin(), v.end(),
    [](int i) {return i % 2 == 1;}); //Noncompliant

    std::vector<int>::iterator first_odd = find_if(v.begin(), v.end(),
    [](int i) {return i % 2 == 1;}); //Noncompliant
}

void func2(std::vector<int>& v)
{
    auto is_odd = [](int i) { return i % 2 == 1;};

    if (none_of(v.begin(), v.end(), is_odd)) {
        //Handle error
    }

    int odds = std::count_if(v.begin(), v.end(), is_odd); //Compliant,
                                                          //reusing named lambda expression

    std::vector<int>::iterator first_odd = find_if(v.begin(),
            v.end(), is_odd); //Compliant, reusing named lambda expression
}
```

In this example, unnamed lambda expression `[](int i) {return i % 2 == 1;}` is reused twice inside `func1`. Polyspace flags the second and third uses of this lambda expression.

The reuse of the lambda expression in `func2` is not flagged because the lambda expression is named (`is_odd`).

## Check Information

**Group:** Expressions
**Category:** Advisory, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-2-1

dynamic_cast should not be used

## Description

### Rule Definition

*dynamic_cast should not be used.*

### Rationale

You use `dynamic_cast` to convert the type of a pointer or reference to a class along the inheritance hierarchy, for instance to convert a pointer to base class into a pointer to a derived class. The conversion incurs an overhead due to the type checking that is performed at run-time. This overhead is unsuitable for the low memory, speed, and predictable performance requirements of real-time systems.

If you cannot avoid dynamic casting in your application, consider using a custom implementation to perform the cast. You might also consider using virtual functions if you are casting to the most derived class, or static polymorphism with overloaded functions and templates. In the latter case, the types are resolved at compile-time which avoids the overhead.

### Polyspace Implementation

Polyspace flags all uses of `dynamic_cast` in your code.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `dynamic_cast`

```
#include<iostream>


using namespace std;

class Base
{
public:
    virtual void f()
    {
        cout << "Using Base class\n";
    }
};

class Derived1 : public Base
{
public:
    virtual void f()
    {
        cout << "Using Derived class\n";
    }
};

class Derived2 : public Derived1
{
public:
    virtual void f()
    {
        cout << "Using Derived2 class\n";
    }
};


int main()
{
    Derived2* ptrd2 = new Derived2;

    Derived1* ptrd1 = dynamic_cast<Derived1*>(ptrd2); // Noncompliant
    ptrd1 -> f();

    Base* ptrb = dynamic_cast<Base*>(ptrd2); // Noncompliant
    ptrb -> f();
}
```

In this example, `Base` and `Derived1` are indirect and direct base classes of `Derived2` respectively. The use of `dynamic_cast` to upcast `ptrd2` from type `Derived2` to `Derived1` then to `Base` is noncompliant. Note that in this case, the use of `dynamic_cast` is not necessary since an upcast can be performed through implicit conversion (`Derived1 * ptr = ptrd2;`).

## Check Information

**Group:** Expressions
**Category:** Advisory, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-2-2

Traditional C-style casts shall not be used

## Description

### Rule Definition

*Traditional C-style casts shall not be used.*

### Rationale

C-style casts are difficult to distinguish in source code. Such casts cannot communicate the goal and necessity of the conversion. Code containing C-style casts is difficult to understand and debug.

Avoid C-style casts. C++ introduces explicit casting operations that are easily identified and clearly communicate the goal and necessity of each cast. Use these casting operations instead.

### Polyspace Implementation

Polyspace flags C-style casts and functional notation casts in your code. Compliant C++ style casting operations include:

- `std::static_cast`
- `std::reinterpret_cast`
- `std::const_cast`
- `std::dynamic_cast`
- `{}` notation casts

The `reference_cast` operation from the Boost library and the `safe_cast` operation from the Microsoft library are also allowed.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid C-Style Casts and Functional Casts

In this example, the function `Foo()` shows several C++ casting operations. These operations are easily identified. They communicate the goal of necessity of the casting operations. For instance, in `Base* obj4 = const_cast<Base*>(&obj3)`, it is clear that the cast operation removes the `const` qualifier from `obj3`. These casting operations are compliant and Polyspace does not flag them.

The function `Bar()` shows several C-style casts and functional casts. These operations are difficult to locate, and their purpose is not communicated. Polyspace flags these casts.

```
#include <cstdint>
class Base
```

```
{
public:
    explicit Base(std::int32_t) {}
    virtual void Fn() noexcept {}
};
class Derived : public Base
{
public:
    void Fn() noexcept override {}
};
class myClass
{
};
std::int32_t G() noexcept
{
    return 7;
}

void Foo() noexcept(false)
{
    Base obj = Base{10};        // Compliant
    const Base obj3(5);
    Base* obj4 = const_cast<Base*>(&obj3);        //Compliant
    Base* obj2;
    myClass* d1 = reinterpret_cast<myClass*>(obj4); // Compliant
    Derived* d2 = dynamic_cast<Derived*>(obj2);       // Compliant
    std::int16_t var1 = 20;
    std::int32_t var2 = static_cast<std::int32_t>(var1);      // Compliant
    std::int32_t var4 = 10;
    float f1 = static_cast<float>(var4);                    // Compliant

    std::int32_t var5 = static_cast<std::int32_t>(f1);      //Compliant

    std::uint32_t var7 = std::uint32_t{0};//Compliant
    static_cast<void>(G());                               //Compliant
}
void Bar(){
    Base obj = Base{1};
    Base* obj2 = (Base*)(&obj); //Noncompliant
    std::int16_t var1 = 20;
    std::int32_t var3 = (std::int32_t)var1;                   // Noncompliant
    float f = float(var3);                                 // Noncompliant
    std::int32_t var6 = (std::int32_t)f;                   // Noncompliant
    std::int32_t var7 = std::int32_t(f);                   // Noncompliant
}
```

### Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-2-3

A cast shall not remove any const or volatile qualification from the type of a pointer or reference

## Description

### Rule Definition

*A cast shall not remove any const or volatile qualification from the type of a pointer or reference.*

### Rationale

Removing the `const` or `volatile` qualification from a pointer or reference might be unexpected. Consider this code:

```
void foo(const char* p){
  *const_cast< char * >( p ) = '\0';
}
```

The function `foo()` accepts a `const` pointer to a `char`. The caller of this functions expects that the parameter `p` remains unchanged. Modifying `p` in `foo()` by converting it to a non-`const` pointer is unexpected. If `*p` dereferences to a `const` character, this modification might lead to unexpected behavior. Avoid casting the `const` or `volatile` away from a pointer or reference.

### Polyspace Implementation

Polyspace raises a violation of this rule if you remove the `const` or `volatile` qualification from the type of a pointer or a reference by using a casting operation.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Casting Away `Const` from Pointers

```
void foo(const char* p){
  *const_cast< char * >( p ) = '\0';//Noncompliant
}

void foo1(volatile char* p){
  (char*) p ;//Noncompliant
}
```

In this example, Polyspace flags the casting operations that cast away the `const` and `volatile` qualifiers from pointers.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-2-4

reinterpret_cast shall not be used

## Description

### Rule Definition

*reinterpret_cast shall not be used.*

### Rationale

`reinterpret_cast` is typically used to explicitly convert between two unrelated data types. For instance, in this example, `reinterpret_cast` converts the type `struct S*` to `int*`:

```
struct S { int x; } s;
int* ptr = reinterpret_cast<int*> (&s);
```

However, it is difficult to use `reinterpret_cast` and not violate type safety. If the result of `reinterpret_cast` is a pointer, it is safe to dereference the pointer only after you cast the pointer back to its original type.

### Polyspace Implementation

The rule checker flags all uses of the `reinterpret_cast` keyword.

If the rule checker flags an use of `reinterpret_cast` that you consider safe, add a comment justifying the result. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `reinterpret_cast` Keyword

```
class A {
    int x;
    int y;
    public:
      void getxy();
};

class B {
    int z;
    public:
      void getz();
};

void func (B* Bptr) {
    A* Aptr = reinterpret_cast<A*>(Bptr); // Noncompliant
}
```

The use of `reinterpret_cast` violates this rule. The result of `reinterpret_cast` is not safe to dereference since `A` and `B` are unrelated classes. Dereferencing `Aptr` as if it were an `A*` pointer can result in illegal memory access.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-2-5

An array or container shall not be accessed beyond its range

## Description

### Rule Definition

*An array or container shall not be accessed beyond its range.*

### Rationale

An array or container accessed beyond its range results in undefined behavior. This rule applies to C-style arrays and all other containers where you access the array or container elements by using an iterator (including pointers) or an index.

To avoid undefined behavior, use appropriate safeguards in your code to make sure that you access the array or container within its range. For example:

- Perform a range check explicitly.
- Use a built-in standard template library (STL) function that performs a range check, such as `array::at()`.
- Use a range-based for loop when appropriate to iterate through the elements of an array or container.

Pointing to one-past the last element of the array or container is well defined, but dereferencing that element is not.

### Polyspace Implementation

Polyspace flags these issues when you enable this rule checker:

- You access a C-style array beyond its range. For instance:

  ```
  #include <iostream>

  void func() {
      int idx, arr[10];

      for (idx = 0; idx < 10; ++idx) {
          arr[idx] = 2 * idx;
      }
      // idx = 10 after for loop
      std::cout << arr[idx]; //Noncompliant
  }
  ```
- You use an unsanitized tainted value as an index to access an element of a C-style array. For instance, in this code snippet, variable `idx` is obtained from the environment list and is used to access an element of array `arr2`. The value of `idx` is not checked and might be outside the range of the array:

  ```
  #include <iostream>

  extern int arr2[50];
  ```

```
void printElem() {
    int idx = strtol(getenv("INDEX"), NULL, 10);
    std::cout << arr2[idx]; //Noncompliant
}
```

Polyspace considers that data from all external sources are tainted. For information about tainted data sources, see “Sources of Tainting in a Polyspace Analysis”.

### Extend Checker

Extend this checker to check for defects caused by specific values and external inputs. For instance:

- A default Bug Finder analysis might not raise a defect when the input values are unknown and only a subset of inputs can cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.
- By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Array Accessed Outside its Range

```
#include <cstdint>

constexpr std::int32_t MAXSIZE = 64;
void foo() {

    int  arr[MAXSIZE]{0};
    int iter;
    for(iter = 0;iter<MAXSIZE;iter++){/**/}
    int val = arr[iter]; //Noncompliant
}



void externalSource(uint8_t externalVar) {

    int8_t idx, arr[10];
    for (idx = 0; idx < 10; ++idx) {
        arr[idx] = idx;
    }
    for (idx = 0; idx < 10; ++idx) {
        if (arr[idx] == arr[externalVar]) { /* Noncompliant, with option
        -consider-analysis-perimeter-as-trust-boundary enabled*/
            // print idx
        }
    }
}
```

In this example:

- In the function `foo()`, after the `for` loop, the value of `iter` is `64`. The array access in `arr[iter]` is out of bound. Polyspace reports a violation on the out of bound array access.
- In the function `externalSource`, the iterator `externalVar` is not checked for validity before it is used to access an element of `arr`. The iterator `externalVar` originated outside of the current module might have a value beyond the bound of `arr`. Polyspace reports a violation when you enable the option `-consider-analysis-perimeter-as-trust-boundary`.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-2-6

The operands of a logical && or || shall be parenthesized if the operands contain binary operators

## Description

### Rule Definition

*The operands of a logical && or || shall be parenthesized if the operands contain binary operators.*

### Rationale

In a logical expression containing binary operators, relying on C++ operator precedence rules results in code that is confusing and difficult to understand. This code might lead to unexpected behavior and bugs that are difficult to resolve. Parenthesizing operands that include binary operators enhances the readability of code, makes code easier to review, and ensures that the operator precedence behavior is as expected.

### Polyspace Implementation

During preprocessing, violations of this rule are detected on the expressions in `#if` directives.

The checker allows exceptions on associativity `(a && b && c)`, `(a || b || c)`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Operands Containing Binary Operators That Are Not Parenthesized

```
#include <cstdint>

void Fn(std::int32_t value) noexcept
{
  if (value > 1 && value < 2) { //Noncompliant
    // do some work
  } else if ((value > 0) && (value < 3)) { //Compliant
    // do some work
  } else if ((value == 0) || value == 3) { //Noncompliant
    // do some work
  } else if ((value < 0) || (value == 4)) { //Compliant
    // do some work
  } else {
    // do some work
  }

  return;
}
```

There are multiple uses of the logicals && and ||. In the first and third logical expressions, there are operands containing binary operators that are not parenthesized. Polyspace flags them as noncompliant with this rule. In the second and fourth logical expressions, all operands containing binary operators are parenthesized. Polyspace does not flag them as noncompliant with this rule.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-3-1

Evaluation of the operand to the typeid operator shall not contain side effects

## Description

### Rule Definition

*Evaluation of the operand to the typeid operator shall not contain side effects.*

### Rationale

The `typeid` operator evaluates its operand only if it is a call to a function that returns a reference to a polymorphic type (a polymorphic type is a class type that contains virtual functions). In all other cases, expressions provided to the `typeid` operator are not evaluated.

For code that is easier to maintain, avoid expressions with side effects altogether when using the `typeid` operator. You or another developer will be saved from tracking down the ingredients of the expression to their definitions and determining if the side effects actually occur.

### Polyspace Implementation

The checker flags `typeid` operators with expressions that have side effects. Function calls are assumed to have side effects.

The checker message states whether the expression is evaluated or ignored. If the expression is evaluated and you want to retain the expression in the `typeid` operation instead of performing the evaluation in a separate statement, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Expressions with Side Effects as `typeid` Operand

```
#include <iostream>
#include <typeinfo>

class Base
{
public:
    virtual ~Base();
};

class Derived: public Base
{
```

```
public:
    ~Derived();
};

Base& getObj();

void main()
{

    Base& b = getObj();

    std::cout << "Dynamic type:" << typeid(getObj()).name(); //Noncompliant
    std::cout << "Dynamic type:" << typeid(b).name(); //Compliant
}
```

The rule is violated when the `typeid` operand involves a function call.

## Check Information

**Group:** Expressions
**Category:** Required, Non-automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-3-2

Null pointers shall not be dereferenced

## Description

### Rule Definition

*Null pointers shall not be dereferenced.*

### Rationale

Dereferencing a null pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

### Polyspace Implementation

The checker flags pointer dereferences where the pointer might be NULL-valued.

If the issue occurs despite an earlier check for `NULL`, look for intermediate events between the check and the subsequent dereference. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Possible Null Pointer Dereference

```
#include <iostream>
#include <cstdint>
#include <cstddef>

class A
{
  public:
    A(std::uint32_t a) : a(a) {}
    std::uint32_t GetA() const noexcept
    {
      return a;
    }
  private:
    std::uint32_t a;
};

std::uint32_t Sum(const A* lhs, const A* rhs)
{
  return lhs->GetA() + rhs->GetA(); //Noncompliant
}

A* getAPtr(void);

int main(void)
{
  A* leftVal = new A(3);
  A* rightVal = getAPtr();

  std::uint32_t sum;

  if(!rightVal)  {
      sum = Sum(leftVal, rightVal);
  }
  else
      sum = 0;

  std::cout << sum << std::endl;
  return 0;
}
```

In this example, the order of the `if` and `else` clause have been switched leading to an accidental null pointer dereference. The variable `rightVal` is checked for NULL and the NULL-valued of `rightVal` is used for the subsequent dereference in the function `Sum`.

## Check Information

**Group:** Expressions
**Category:** Required, Partially automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-3-3

Pointers to incomplete class types shall not be deleted

## Description

### Rule Definition

*Pointers to incomplete class types shall not be deleted.*

### Rationale

When you delete a pointer to an incomplete class, it is not possible to call any nontrivial destructor that the class might have. If the destructor performs cleanup activities such as memory deallocation, these activities do not happen.

A similar problem happens, for instance, when you downcast to a pointer to an incomplete class (downcasting is casting from a pointer to a base class to a pointer to a derived class). At the point of downcasting, the relationship between the base and derived class is not known. In particular, if the derived class inherits from multiple classes, at the point of downcasting, this information is not available. The downcasting cannot make the necessary adjustments for multiple inheritance and the resulting pointer cannot be dereferenced.

### Polyspace Implementation

The check raises a defect when you delete or cast to a pointer to an incomplete class. An incomplete class is one whose definition is not visible at the point where the class is used.

For instance, the definition of class `Body` is not visible when the `delete` operator is called on a pointer to `Body`:

```
class Handle {
  class Body *impl;
public:
  ~Handle() { delete impl; }
  // ...
};
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Deletion of Pointer to Incomplete Class

```
class Handle {
  class Body *impl;
public:
  ~Handle() { delete impl; }
  // ...
};
```

In this example, the definition of class Body is not visible when the pointer to Body is deleted.

#### Correction — Define Class Before Deletion

One possible correction is to make sure that the class definition is visible when a pointer to the class is deleted.

```
class Handle {
  class Body *impl;
public:
  ~Handle();
  // ...
};

// Elsewhere
class Body { /* ... */ };

Handle::~Handle() {
  delete impl;
}
```

#### Correction — Use std::shared_ptr

Another possible correction is to use the std::shared_ptr type instead of a regular pointer.

```
#include <memory>

class Handle {
  std::shared_ptr<class Body> impl;
  public:
    Handle();
    ~Handle() {}
    // ...
};
```

### Downcasting to Pointer to Incomplete Class

File1.h:

```
class Base {
protected:
  double var;
public:
  Base() : var(1.0) {}
  virtual void do_something();
  virtual ~Base();
};
```

File2.h:

```
void funcprint(class Derived *);
class Base *get_derived();
```

File1.cpp:

```
#include "File1.h"
#include "File2.h"

void getandprint() {
  Base *v = get_derived();
  funcprint(reinterpret_cast<class Derived *>(v));
}
```

`File2.cpp`:

```
#include "File2.h"
#include "File1.h"
#include <iostream>

class Base2 {
protected:
  short var2;
public:
  Base2() : var2(12) {}
};

class Derived : public Base2, public Base {
  float var_derived;
public:
    Derived() : Base2(), Base(), var_derived(1.2f) {}
    void do_something()
    {
        std::cout << "var_derived: "
                  << var_derived << ", var : " << var
                  << ", var2: " << var2 << std::endl;
    }
 };

void funcprint(Derived *d) {
  d->do_something();
}

Base *get_derived() {
  return new Derived;
}
```

In this example, the definition of class `Derived` is not visible in `File1.cpp` when a `Base*` pointer to downcast to a `Derived*` pointer.

In `File2.cpp`, class `Derived` derives from two classes, `Base` and `Base2`. This information about multiple inheritance is not available at the point of downcasting in `File1.cpp`. The result of downcasting is passed to the function `funcprint` and dereferenced in the body of `funcprint`. Because the downcasting was done with incomplete information, the dereference can be invalid.

#### Correction — Define Class Before Downcasting

One possible correction is to define the class `Derived` before downcasting a `Base*` pointer to a `Derived*` pointer.

In this corrected example, the downcasting is done in `File2.cpp` in the body of `funcprint` at a point where the definition of class `Derived` is visible. The downcasting is not done in `File1.cpp` where the definition of `Derived` is not visible. The changes from the previous incorrect example are highlighted.

`File1_corr.h`:

```
class Base {
protected:
  double var;
public:
  Base() : var(1.0) {}
  virtual void do_something();
  virtual ~Base();
};
```

File2_corr.h:

```
void funcprint(class Base *);
class Base *get_derived();
```

File1.cpp:

```
#include "File1_corr.h"
#include "File2_corr.h"

void getandprint() {
  Base *v = get_derived();
  funcprint(v);
}
```

File2.cpp:

```
#include "File2_corr.h"
#include "File1_corr.h"
#include <iostream>

class Base2 {
protected:
  short var2;
public:
  Base2() : var2(12) {}
};

class Derived : public Base2, public Base {
  float var_derived;

public:
    Derived() : Base2(), Base(), var_derived(1.2f) {}
    void do_something()
    {
        std::cout << "var_derived: "
                  << var_derived << ", var : " << var
                  << ", var2: " << var2 << std::endl;
    }
};

void funcprint(Base *d) {
  Derived *temp = dynamic_cast<Derived*>(d);
  if(temp)  {
     d->do_something();
  }
  else {
      //Handle error
  }
}
```

```
Base *get_derived() {
  return new Derived;
}
```

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A5-5-1

A pointer to member shall not access non-existent class members

## Description

### Rule Definition

*A pointer to member shall not access non-existent class members.*

### Rationale

You use pointer-to-member operators `.*` and `->*` to refer to a non-static class member of the first operand object. For instance, in this code snippet, the call `(obj.*ptrToMember)();` is equivalent to `obj.f();`. Both calls refer to member function `f()` of class `myObj`:

```
class myObj {
  public:
    void f();
};

void func() {
    myObj obj;
    void (myObj::*ptrToMember)() = &myObj::f;

    (obj.*ptrToMember)(); // Equivalent to obj.f();
}
```

The use of pointer-to-member operators results in undefined behavior in these cases:

- The dynamic type of the first operand does not contain the member referred to by the second operand.
- The second operand pointer is null.

### Polyspace Implementation

Polyspace flags the use of pointer-to-member operators in these instances:

- You cast a pointer-to-member type to another pointer-to-member type which does not contain the class member pointed to. For instance, in this code snippet, pointer-to-member `ptrToMember` is obtained from `myObj::f` but is then upcast to `baseObj::*` which does not contain a method `f`.

  ```
  class baseObj {
    public:
      virtual ~baseObj() = default;
  };

  class myObj : public baseObj {
    public:
      void f();
  };
  ```

```
void func() {
    baseObj* foo = new baseObj();
    void (baseObj::*ptrToMember)() =
        static_cast<void (baseObj::*)()>(&myObj::f); //Noncompliant
    (foo->*ptrToMember)();
}
```

Polyspace does not flag the casting operation if the type being cast to inherits from the other type. For example, in the preceding code, if `baseObj` inherits from `myObj`.

- The second operand of the pointer-to-member operator is a null pointer. For instance, in this code snippet, `ptrToMember` is declared but not initialized and defaults to a null pointer.

```
class baseObj {
  public:
    virtual ~baseObj() = default;
};
static void (baseObj::*ptrToMember)(); //Not initialized

void func() {
    baseObj* foo = new baseObj();

    (foo->*ptrToMember)(); //Noncompliant
}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Pointer-to-Member Accesses Non-Existent Class Member

```
class Base {
  public:
    void f() {
    }
};

class Derived : public Base {
  public:
    int myVar;
};

void func() {
    Base myObj;
    auto ptrToMethod =
        static_cast<void (Base::*)()>(&Derived::f); // Compliant
    auto ptrToMember =
        static_cast<int(Base::*)>(&Derived::myVar); // Noncompliant

    (myObj.*ptrToMethod)(); // OK, equivalent to myObj.f()
    myObj.*ptrToMember;     // Undefined behavior
}
```

In this example, Polyspace flags the casting of `ptrToMember`, which is obtained from `Derived::myVar`, to `Base::*` because this class does not contain a member `myVar`. If you execute the code, the call `myObj.*ptrToMember;` results in undefined behavior.

Polyspace does not flag the casting of `ptrToMethod` to `Base::*` because class `Derived` inherits method `f` from `Base`.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-6-1

The right hand operand of the integer division or remainder operators shall not be equal to zero

## Description

### Rule Definition

*The right hand operand of the integer division or remainder operators shall not be equal to zero.*

### Rationale

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

### Polyspace Implementation

The checker raises a defect when:

- The denominator of a division or modulo operation can be a zero-valued integer.
- There are division operations where one or both of the integer operands is from an unsecure source.
- There are modulo operations with one or more tainted operands.

### Extend Checker

Extend this checker to check for defects caused by specific values and external inputs. For instance:

- A default Bug Finder analysis might not raise a defect when the input values are unknown and only a subset of inputs can cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.
- By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

#### Correction — Check Before Division

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

#### Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2;
    int result = 0;

    result = num/denom;

    return result;
}
```

### Modulo Operation with Zero

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        arr[i] = input % i;
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

In this example, Polyspace flags the modulo operation as a division by zero. Because modulo is inherently a division operation, the divisor (right hand argument) cannot be zero. The modulo operation uses the `for` loop index as the divisor. However, the `for` loop starts at zero, which cannot be an iterator.

#### Correction — Check Divisor Before Operation

One possible correction is checking the divisor before the modulo operation. In this example, see if the index `i` is zero before the modulo operation.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        if(i != 0)
        {
             arr[i] = input % i;
        }
        else
        {
             arr[i] = input;
        }
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

#### Correction — Change Divisor

Another possible correction is changing the divisor to a nonzero integer. In this example, add one to the index before the `%` operation to avoid dividing by zero.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
         arr[i] = input % (i+1);
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

### Division of Function Arguments

```
#include <limits.h>
#include <stdio.h>

extern void print_int(int);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    int r =  num/denum; //Noncompliant
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

#### Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include <limits.h>
#include <stdio.h>

extern void print_long(long);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    long res= 0;
    if (denum!=0 && !(num==INT_MIN && denum==-1)) {
        res = num/denum;
    }
    print_long(res);
    return res;
}
```

### Modulo with User Input

```
#include <stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem =  128%userden; //Noncompliant
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using a user input. The input is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

#### Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
#include<stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem = 0;
    if (userden > 0 ) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-10-1

A pointer to member virtual function shall only be tested for equality with null-pointer-constant

## Description

### Rule Definition

*A pointer to member virtual function shall only be tested for equality with null-pointer-constant.*

### Rationale

A call to a member virtual function is resolved by the compiler at run-time to the most derived version of the function. If you use the equality operators (==) or (`!=`) to compare anything other than the null-pointer constant `nullptr` with a pointer to a member virtual function, the result is unspecified.

### Polyspace Implementation

Polyspace flags any (==) or (`!=`) comparison where one operand is a pointer to a member virtual function and the other operand is not `nullptr`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Equality Comparison Between Pointer to Virtual Member Function and Non-`nullptr` Operand

```
class Base
{
public:
    virtual void f();
    void g();
};

template<typename T>
class Derived : public Base
{
public:
    void f();
};

void f()
{

    bool b = (&Derived<int>::f == &Derived<int>::f); // Noncompliant

    void (Derived<float>::* p)() = &Derived<float>::f;
    bool b1 = (&Derived<float>::f == p); // Noncompliant
    bool b2 = (p == p); // Noncompliant
    bool b3 = (p == nullptr); // Compliant

    void (Base::* q)() = &Base::g;
    bool b4 = (q == q); // Compliant

    void (Base::* r)() = &Base::f;
    bool b5 = (q == r); // Noncompliant

}
```

In this example, the result of the comparison in boolean `b` is non-compliant because the operands point to a member virtual function. Similarly, pointers `p` and `r` are pointers to member declarators that point to a member virtual function and Polyspace flags their use in equality comparison operations, except for the comparison of `p` to `nullptr`.

### Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2020b**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A5-16-1

The ternary conditional operator shall not be used as a sub-expression

## Description

### Rule Definition

*The ternary conditional operator shall not be used as a sub-expression.*

### Rationale

A ternary conditional operator used as a subexpression makes the full expression less readable and difficult to maintain. It is often visually clearer if you assign the result of a ternary operator to a variable and then use the variable in subsequent operations.

### Polyspace Implementation

The checker flags uses of the ternary conditional operator in subexpressions with some exceptions. Exceptions include uses of the operator when:

- The result is assigned to a variable.
- The result is used as a function argument or returned from a function.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Ternary Operators as Sub-expressions

```
#include <cstdint>
const int ULIM = 100000;

std::int32_t foo(int32_t x) {
    int ret;
    ret = (x <= 0? 0: (x >= ULIM? 0 : x)); //Noncompliant
    return ret;
}

std::int32_t bar(int32_t x) {
    int ret, retInterim;
    retInterim = x >= ULIM? 0 : x; //Compliant
    ret = retInterim <= 0? 0 : retInterim; //Compliant
    return ret;
}
```

In this example, in `foo`, a ternary conditional operation is chained with a second operation to return the value 0 if `x` is in the range `[0, ULIM]` and return `x` otherwise. The ternary operation comparing `x` with `ULIM` is a sub-expression in the full chain and violates the rule.

In `bar`, each ternary conditional operation is written in a separate step and does not violate the rule. Alternatively, the same algorithm can be implemented by combining the conditions with the boolean AND operator and using a single ternary conditional operation.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A6-2-1

Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects

## Description

### Rule Definition

*Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects.*

### Rationale

When you use move and copy assignment operators, you expect that the operation moves or copies a source object to a target object without producing any side effects. If move or copy assignment operators of a class produce side effects, the invariant of an object can change during move or copy assignments. Consider this code where multiple objects of class `C` are copy-assigned to each other.

```
class C{
    //...
    C& operator=(const C& oth) {
        value = oth.value / 2;
        return *this;
    }
public:
    int value;
};

int main(){
    C a, b, c;
    a.value = 48;
    // …
    b = a; // b.m = 24
    c = b; // c.m = 12
    a = c; // a.m = 6
}
```

During each copy assignment, the `value` in the target object becomes half of the `value` in the source object. After three successive copy assignment operations, `a.value` becomes `6`, which is unexpected. Algorithms expect move and copy assignment operators that do not change the object invariant. If move or copy assignment operators of a class have side effects that change the object invariant, using algorithm libraries such as the standard template library (STL) can produce unexpected results.

Because you use move and copy assignments many times in a code, any side effect producing code can make the code slower and resource intensive. In a move assignment operator, code that produce side effects can also cause the compiler to use copy operation with every assignment, which is inefficient.

To maintain optimum and reliable performance during move and copy assignment, perform only these operations in move and copy assignment operators:

- Copy or move data members and base classes.

- Return the pointer *this.
- If possible, set the moved-from object to a valid state.

Avoid superfluous code that add unrelated side effects or performance overhead.

### Polyspace Implementation

In the body of a copy or move assignment operator, Polyspace does not flag these operations:

- Copy or move assignments.
- Relational or comparison operations.
- Modification of the source object in a move operation.
- Calls to the function std::swap or equivalent user-defined noexcept swap functions. Polyspace identifies functions that these signatures as swap functions: void T::swap(T&) or void [N::]swap(T&, T&). The first signature represents a member function of class T that takes one argument. The second signature represents a nonmember or static function in the namespace N that takes two arguments. The name swap can be case-insensitive and prefixed or postfixed by underscores.
- Assignment and modification of static variables.

Polyspace flags any other operations in a copy or move assignment operator as unwanted side effect. For instance, a call to a user-defined swap function is considered an unwanted side effect if the swap function is not noexcept. For a similar rule on copy and move constructor, see AUTOSAR C++14 Rule A12-8-1.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Move and Copy Assignment Operators with Side Effects

This code shows how Polyspace flags move and copy assignment operators that have side effects.

```
#include<cstdint>
#include<iostream>
class B
{
public:
    B() : ptr(0) {}
    B& operator=(B&& oth) //Noncompliant
    {

        if(&oth == this) {
            return *this;
        }
        ptr = std::move(oth.ptr);
        std::cout<<"Moved";
        return *this;
    }

private:
    std::int32_t* ptr;
};
class C
{
public:
    C(int t=0) : x(t) {}
    C& operator=(const C& oth)  // Noncompliant
    {
        if(&oth == this) {
            return *this;
        }
        x = oth.x % 2;         // This operation produces side-effect
        count++; //Not a side effect
        return *this;
    }

private:
    std::int32_t x;
    static std::int32_t count;

};
class D
{
public:
    D(const D&) = default;
    D(D&&) = default;

    D& operator=(const D& oth) & {     // Noncompliant
        D tmp(oth);
        swap(tmp);
        return *this;
    }

    // Member function swap
    void swap(D& rhs)  {
        //...
    }

private:
    std::int32_t x;
};
```

- As a side effect, the move assignment operator of class B prints a string into the output stream. This side effect adds performance overhead to the move operation. If this statement `std::cout<<"Moved"` causes an exception, code execution can unexpectedly stop. Polyspace flags the move assignment operator and highlights the statement.
- The copy assignment operator of C modifies the data member `x` of the source object. This side effect adds performance overhead. Unexpected change to data members during move and copy operations can make the code incompatible with the standard template library and introduce errors during development. Polyspace flags the copy assignment operator and highlights the statement `x = oth.x % 2`. Incrementing the static variable `count` is not a side effect.
- The copy assignment operator of the class `D` calls a user-defined swap function called `_swap_`. This swap function is not `noexcept`. If an exception is raised from `_swap_`, the exception is an unexpected side effect of the copy assignment operator. Polyspace flags the copy constructor as noncompliant with this rule. Use user-defined swap function that are `noexcept`.

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A6-2-2

Expression statements shall not be explicit calls to constructors of temporary objects only

## Description

### Rule Definition

*Expression statements shall not be explicit calls to constructors of temporary objects only.*

### Rationale

Objects that the compiler creates for a short duration, and then deletes, are temporary objects. The compiler might create temporary objects for specific purposes, such as:

- Initializing references
- Storing values returned by functions
- Type casting
- Exception handling

Temporary objects are destroyed once the expression that requires their construction is completely evaluated. For instance, in evaluating the expression `sum = a*b+c`, the compiler creates two temporary objects to store the results of the multiplication and addition operations. After the expression is evaluated, both temporary objects are destroyed. Their scope is limited to the expression statement.

If an expression is an explicit call to a constructor omitting the object name, the compiler creates a temporary object which is immediately destroyed. Such an explicit call to a constructor might indicate that:

- You inadvertently omitted the object name.
- You expected the unnamed variable to remain in scope up to the end of the declaration block.

Consider this code snippet where a `lock_guard` object is created.

```
void foo(){
std::mutex mymutex;
std::mutex mymutex2;
std::lock_guard<std::mutex> lock{mymutex};
std::lock_guard<std::mutex> {mymutex2};
//...
}
```

The first declaration creates a `lock_guard` object named `lock`. The object `lock` protects `mymutex` from concurrent access by multiple thread until the end of the current block. The second declaration attempts a similar protection for `mymutex2`. Because the `lock_guard` object in this case is not named, it is destroyed immediately after the declaration statement. Perhaps inadvertently, `mymutex2` remains unprotected from concurrency issues.

Avoid expression statements that are only an explicit call to a constructor. To implement the Resource Acquisition Is Initialization (RAII) pattern, use named objects.

### Polyspace Implementation

Polyspace flags any expression statement that constructs an unnamed object and does not use it. You can construct unnamed temporary objects when you use the objects within the declaration expression statement. For example, a temporary object that is used as a function return or on the right-hand side of an assignment is compliant with this rule.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Statements That Are Only Explicit Calls To Constructors

This code shows how Polyspace flags statements that are only explicit calls to a constructor.

```
#include <cstdint>
#include <fstream>
#include <string>
class MyException {
    MyException(const std::string &);
};
void with_exception() {
    MyException("Exception");              //Noncompliant
    throw MyException("Exception");        //Compliant

};
```

Polyspace flags an expression statement that constructs an unnamed temporary object and does not use it. If you use the temporary object in the statement, then the statement is compliant with the rule. For example, the statement `MyException("Exception");` is flagged because the unnamed object created by the explicit call to the constructor `MyException()` is not used in the statement. The statement `throw MyException("Exception");` is not flagged because the unnamed object is used as an argument to `throw`.

### Avoid Unnamed `lock_guard` Objects

Compilers destroy an unnamed `lock_guard` object immediately after its declaration statement. Unnamed `lock_guard` objects cannot protect `mutex` objects from concurrency issues. Polyspace flags a statement when it declares an unnamed `lock_guard` object. Consider this code:

```
#include <cstdint>
#include <mutex>
class A {
public:
    void SetValue1(std::int32_t value) {
        std::lock_guard<std::mutex> {mutex1}; //Noncompliant
        private_value = value;
    }

    void SetValue2(std::int32_t value) {
        std::lock_guard<std::mutex> lock{mutex2}; //Compliant
        private_value = value;
    }
private:
    mutable std::mutex mutex1;
    mutable std::mutex mutex2;
    std::int32_t private_value;
};
```

- The statement `std::lock_guard<std::mutex> {mutex1};` declares an unnamed `lock_guard` object. Polyspace flags the statement.
- The statement `std::lock_guard<std::mutex> lock{mutex2};` is not flagged because the `lock_guard` object is named.

## Check Information

**Group:** Statements
**Category:** Required, Automated

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A6-4-1

A `switch` statement shall have at least two case-clauses, distinct from the default label

## Description

### Rule Definition

*A `switch` statement shall have at least two case-clauses, distinct from the default label.*

### Rationale

A `switch` statement with no case-clauses is redundant. Use an `if` statement to better express a `switch` statement with a single case-clause.

### Polyspace Implementation

This rule checker reports a violation for `switch` statements that contain less than two case-clauses in addition to the default-clause. This includes situations where no case-clause exists.

```
    switch (x) {    //Noncompliant
        default:
                   //...
            break;
    }
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Example

### Use at Least Two Case-Clauses in `switch` Statements

Because this `switch` statement contains a single case-clause alongside the default-clause, Polyspace flags the `switch` statement as noncompliant.

```
void example(int x)
{
    switch (x) {    //Noncompliant
        case 0:
            break;
        default:
            break;
    }
}
```

You can express the same code in an `if` statement form to avoid a rule violation.

```
void example(int x)
{
    if (x == 0)
    {
        //...
    }
    else
    {
        //...
    }
}
```

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A6-5-1

A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used

## Description

### Rule Definition

*A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used.*

### Rationale

If you loop through all the elements of a container and you use the loop counter variable to access only the value of the elements, a `for` loop is slower, less readable, and more difficult to maintain than an equivalent range-based `for` loop.

### Polyspace Implementation

Polyspace flags the use of `for` loops that loop through all elements of a container or a C-style array when both of these conditions are true:

- The `for` loop has only one loop counter variable. For instance, Polyspace does not flag this `for` loop:

  ```
  for(int i=0, j=0; i < 10; ++i, ++j){ //Compliant
  //...
  }
  ```
- The `for` loop uses the loop counter variable only to access the elements of the container or the C-style array. For instance, in this code snippet, Polyspace flags the first `for` loop because the array counter "`i`" is used only the access elements of `myArray` which are then assigned a value. Use a range-based `for` loop instead to perform an equivalent operation.

  Polyspace does not flag the second `for` loop because the array counter is used to access the array elements and is assigned to the element.

  ```
  #include <iostream>

  int myArray[10];

  void cArray()
  {
      //First for loop: Loop counter 'i' used only to access elements
      //of myArray which are then assigned a value
      for (int i = 0; i < 10; ++i)
      { //Non-compliant
          myArray[i] = 0;
      }
      //Range-based for loop - equivalent to first for loop
      for (int idx : myArray)
      {
          myArray[idx] = 0; // compliant
      }
  ```

```
    //Second for loop: Loop counter assigned to elements of myArray
    for (int i = 0; i < 10; ++i)
    { //Compliant
        myArray[i] = i;
    }
}
```

Polyspace does not flag violations of this rule in these scenarios:

- You iterate through a user-defined container. For instance, this `for` loop is compliant.

  ```
  template<typename T>
  class customContainer {
    typedef T* iterator;
    public:
      iterator begin();
      iterator end();
  };

  void func() {
    customContainer<int> myContainer;

    for(auto it = myContainer.begin(); it != myContainer.end(); it++) { //Compliant
      std::cout << *it;
    }
  }
  ```

  Polyspace flags violations of the rule only for C-style arrays and Standard Template Library containers.

- You use reverse iterators to loop through the elements of the C-style array or container. For instance, in this code snippet, there is no violation of the rule because you cannot use a range-based `for` loop to perform an equivalent operation.

  ```
  #include <iostream>
  #include <vector>
  #include <cstdint>

  std::vector<std::uint32_t> myVector(10);

  void myContainer()
  {
      //loop uses reverse iteration
      for (auto it = myVector.rbegin(); it != myVector.rend(); ++it) { //Compliant

          std::cout << *it;
      }
  }
  ```

- If you loop through arrays:

  - You loop through the elements of multiple arrays or you loop through the elements of a multidimensional array.
  - The array size is unknown.

  For instance, Polyspace considers these `for` loops compliant:

  ```
  int myArray[10];
  int myOtherArray[10];
  int multiArray[10][10];

  void cArray()
  {
      //loop through multiple array
      for (int i = 0; i < 10; ++i) { //Compliant
          myArray[i] = 0;
  ```

```
            myOtherArray[i] = 0;
        }
        //loop through 2-dimensional array
        for (int i = 0; i < 10; ++i) { //Compliant
            multiArray[i][i] = 0;
        }

    }

    void unknownSize(int someArray[])
    {
        //loop through array of unknown size
        for (int i = 0; i < 10; ++i) { //Compliant
            someArray[i] = 0;
        }
    }
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Inefficient Use of a for Loop

```
#include <iostream>
#include <vector>
#include <cstdint>

std::vector<std::uint32_t> myVector(10);

void myContainer()
{
    for (auto it = myVector.begin(); it != myVector.end(); ++it) { //Non-compliant
        std::cout << *it;
    }

    for (auto it : myVector) { //Compliant
        std::cout << it;
    }
}
```

In this example, Polyspace flags the `for` loop, which iterates through all the elements of vector `myVector` and uses the loop counter `it` to only print the value of each element. The range-based for loop `for (auto it: myVector)` performs an equivalent operation, uses less code and makes your code run more efficiently, more readable, and easier to maintain.

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**
“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A6-5-2

A `for` loop shall contain a single loop-counter which shall not have floating-point type

## Description

### Rule Definition

*A `for` loop shall contain a single loop-counter which shall not have floating-point type.*

### Rationale

In situations where your code has a `for` loop without a loop counter, replace the `for` loop with a `while` loop.

In floating-point arithmetic operations, floating-point types are rounded to fit into a finite representation. This introduces rounding errors which can cause unexpected results when loop-counter comparisons occur.

### Polyspace Implementation

This rule checker reports a rule violation in these situations:

- The `for` loop index has a floating-point type.
- You increment more than one loop counter in the `for` loop increment statement.

  For instance:

  ```
  for(i=0, j=0; i<10 && j < 10;i++, j++) {}
  ```
- You do not increment a loop counter in the `for` loop increment statement.

  For instance:

  ```
  for(i=0; i<10;) {}
  ```

  Even if you increment the loop counter in the loop body, Polyspace still reports a violation. The rule is based on MISRA C++ rule 6-5-1. According to the MISRA C++ specifications, a loop counter is one that is initialized in or prior to the loop expression, acts as an operand to a relational operator in the loop expression, and is modified in the loop expression. If the increment statement in the loop expression is missing, the rule checker cannot find the loop counter modification and considers the loop counter not present.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Example

### Use Single Loop Counter Only

Because the code increments both loop counters, `x` and `i`, this example is noncompliant.

```
#include <iostream>

int main()
{
    int x = 0;
    for (int i = 0; i >= x; i = x++)   // Non-compliant
    {
        //...
    }
}
```

#### Avoid Using Floating-Point Numbers as Loop Counters

In this noncompliant example, because floating-point calculations contain rounding errors, the `for` loop executes only nine times. This is due to the value of `x` being larger than `1.0f` on the tenth loop.

```
#include <iostream>

int main()
{
    for (float x = 0.1f; x <= 1.0f; x += 0.1f) {   //Noncompliant
        std::cout << x << std::endl;
    }
}
```

### Check Information

**Group:** Statements
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A6-5-3

Do statements should not be used

## Description

### Rule Definition

*Do statements should not be used.*

### Rationale

A do statement can introduce bugs in your code because its termination condition is checked after executing the code block. Consider this code where an array is accessed by using a pointer in a do-while loop.

```
int* array;
//...
do {
cout<<*array;
--array;
} while (array != nullptr);
```

Because the termination condition is checked after executing the block of code, this code might dereference an invalid or null pointer, which can unexpectedly terminate code execution during run time. The code is also hard to read because the condition for executing the block is at the end of the block where it can be easily missed.

Avoid do statements in your code. You can use do statements to write function-like macros.

### Polyspace Implementation

Polyspace flags all do statements, except those located in macros.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid do Statements

This example shows how Polyspace flags do statements outside of a macro. This code uses a pointer within a do-while loop. The terminating condition is that the pointer is not a null pointer.

```
#include<cstdint>
struct P
{
    int val;
    struct P* next;
};
```

```
void psKO(P*p)
{
    do              // Noncompliant
    {
        p = p->next;
    } while(p!=nullptr);
}
```

This code can dereference a null pointer because the terminating condition is checked after executing the do block. The code is also difficult to read because the terminating condition is placed at the end of the block. Polyspace flags the do statement.

### do Statements in Macros

This example shows how Polyspace treats do statements in a macro. Consider this code where two macros, SWAP and SWAP2 are implemented. SWAP uses a do statement while SWAP2 does not.

```
#include <cstdint>
//Compliant by exception
#define SWAP(a, b) \
        do                          \
        {                           \
            decltype(a) tmp = (a); \
           (a) = (b);              \
           (b) = tmp;              \
       } while (0)

#define SWAP2(a, b)        \
       decltype(a) tmp = (a); \
       (a) = (b);             \
       (b) = tmp;

int main(void)
{
  uint8_t a = 24;
  uint8_t b = 12;

  if (a > 12)
//  SWAP2(a, b);  // Compilation Error
  SWAP(a, b);
return 0;
}
```

The two macros are intended to be invoked like functions. You cannot use SWAP2 as a function-like macro in the if block because after expansion, only the first expression statement of SWAP2 remains within the if block. This breakup of the macro changes its meaning, and in this case, causes a compilation error. A solution to this issue is to enclose the macro in a do-while block and put the terminating condition as false. Such enclosed macros cannot be broken up and can be invoked as functions. Polyspace does not flag do statements in macros.

## Check Information

**Group:** Statements
**Category:** Advisory, Automated

## Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A6-5-4

For-init-statement and expression should not perform actions other than loop-counter initialization and modification

## Description

### Rule Definition

*For-init-statement and expression should not perform actions other than loop-counter initialization and modification.*

### Rationale

For-init-statements and expressions are easier to read, understand, and maintain when they contain only the loop-counter initialization and modification.

### Polyspace Implementation

This checker flags the following situations

- Reports if `loop` parameter cannot be determined. Assumes JSF C++ Rule 200 is not violated. The `loop variable` parameter is assumed to be a variable.
- Assumes 1 loop parameter (see JSF C++ Rule 198), with non class type. JSF C++ Rule 200 must not be violated for this rule to be reported.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Noncompliant Actions in For-Init-Statement

Because the `for` loop contains multiple loop parameters, Polyspace flags the `for` loop as noncompliant.

```
#include <cstdint>

void example()
{
    for (std::int32_t x, y = 0; x < y; x++)  //Noncompliant
    {
        //...
    }
}
```

## Check Information

**Group:** Statements
**Category:** Advisory, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A6-6-1

The `goto` statement shall not be used

## Description

### Rule Definition

*The `goto` statement shall not be used.*

### Rationale

Unrestricted use of `goto` statements increases the complexity of the logic of a program and makes the code difficult to understand. Additionally, `goto` statements can introduce memory leaks or incorrectly release resources.

### Polyspace Implementation

Polyspace reports a rule violation when you use a `goto` statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `goto` Statements

Polyspace reports a rule violation every time this code uses a `goto` statement.

```
void foo(void)
{
    int i = 0, result = 0;

label1:
    for (i; i < 5; i++) {
        if (i > 2) goto label2;     // Non-compliant
    }

label2: {
        result++;
        goto label1;                // Non-compliant
    }
}
```

## Check Information

**Group:** Statements
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A7-1-1

Constexpr or const specifiers shall be used for immutable data declaration

## Description

### Rule Definition

*Constexpr or const specifiers shall be used for immutable data declaration.*

### Rationale

Declaring a variable `const` or `constexpr` reduces the chances that you modify the variable by accident. In addition, compilers can perform various optimizations on `const` and `constexpr` variables to improve run-time performance.

### Polyspace Implementation

The checker flags:

- Function parameters or local variables that are not const-qualified but never modified in the function body.
- Pointers that are not const-qualified but point to the same location during its lifetime.

Function parameters of integer, float, enum, and Boolean types are not flagged.

If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. These variables are not flagged.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unmodified Local Variable

```
#include <cstddef>
bool Status;
extern int setStatus(); //sets Status; returns -1 if fails

char getNthChar(const char* str, int N){ //noncompliant
    int index=0;
    int status = setStatus(); //noncompliant
    while(*(str+index)!='\0'){
        if(index==N)
            return *(str+N);
        ++index;
    }

    return '\0';
}
```

```
char getNthChar_const_safe(const char* const str, int N){
    int index=0;
    const int status = setStatus();
    while(*(str+index)!='\0'){
        if(index==N)
            return *(str+N);
        ++index;
    }

    return '\0';
}
```

In the function `getNthChar()`, the C-string `str` is passed as a `const char*` parameter, meaning that the string `*str` is `const`. Because the pointer `str` does not change value, the pointer itself must be `const` qualified, as shown in the function `getNthChar_const_safe`. Polyspace flags the parameter `str`.

Because the local integer `status` is not modified in `getNthChar()`, it must be declared `const`, as shown in `getNthChar_const_safe`. Polyspace flags the declaration.

### Check Information

**Group:** Declaration
**Category:** Required, Automated

## Version History

**Introduced in R2020b**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A7-1-2

The constexpr specifier shall be used for values that can be determined at compile time

## Description

### Rule Definition

*The constexpr specifier shall be used for values that can be determined at compile time.*

### Rationale

If a variable value is computed from an expression that involves compile-time constants only, using `constexpr` before the variable definition, like this:

```
constexpr double eValSquared = 2.718*2.718;
```

ensures that the expression is evaluated at compile time. The compile-time evaluation saves on run-time overheads. Sometimes, the performance gains at run time can be significant.

If the expression cannot be evaluated at compile time, the `constexpr` keyword ensures that you get a compilation error. You can then fix the underlying issue if possible.

Note that the `const` keyword does not guarantee compile-time evaluation. The `const` keyword simply forbids direct modification of the variable value after initialization. Depending on how the variable is initialized, the initialization can happen at compile time or run time.

### Polyspace Implementation

The checker flags a local variable definition without the `constexpr` specifier if the variable is initialized with one of the following and not modified subsequently in the code:

- A compile-time constant, for instance, a literal value.
- An expression involving compile-time constants only.
- Calls to a function with compile-time constants as parameters, provided the function is itself `constexpr` or the function contains only a return statement involving its parameters.
- A constructor call with a compile-time constant, provided all member functions of the class including the constructor are themselves `constexpr`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Potential `constexpr` Variables

```
double squareIfPositive(double val) {
    return val > 0? (val * val): 0;
}
```

```
constexpr double square(double val) {
    return val > 0? (val * val): 0;
}

void initialize(void) {
    double eVal = 2.718; //Noncompliant
    double eValSquare = squareIfPositive(2.718); //Noncompliant
    const double eValCubed = 2.718 * 2.718 * 2.718; //Noncompliant

    constexpr double eValSquareAnother = square(2.718); //Compliant
}
```

In this example, the first three variable definitions in the `initialize` function are noncompliant because the variables are initialized with expressions involving literal values and the `constexpr` keyword is omitted.

### Check Information

**Group:** Declaration
**Category:** Required, Automated

## Version History

**Introduced in R2020b**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A7-1-3

CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name

## Description

### Rule Definition

*CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.*

### Rationale

Suppose a `typedef` or `using` statement defines a pointer type. For instance:

```
using IntPtr = std::int32_t*;
```

A `const`-qualification of the type written as:

```
const IntPtr ptr = &someValue;
```

Results in this expansion:

```
const (std::int32_t*) ptr = &someValue;
```

In this expression, `ptr` is a constant pointer, which cannot be reassigned to another memory location. However, a developer or reviewer might expect this expansion:

```
(const std::intr32_t) *ptr = &someValue;
```

In this expression, `ptr` is a pointer to a constant, which means that the contents of the location that `ptr` points to, or `*ptr`, cannot be changed.

To avoid this confusion, place a `const` or `volatile` qualifier to the right of a data type defined through `typedef` or `using`. For instance:

```
IntPtr const ptr = &someValue;
```

The only possible expansion of this expression is:

```
std::intr32_t const *ptr = &someValue;
```

which makes `ptr` a constant pointer.

### Polyspace Implementation

The checker reports a violation if `const` or `volatile` qualifiers are placed on the left side of data types defined through `typedef` or `using` statements.

The checker reports violations on both pointer and nonpointer data types. The checker does not report a violation on `typedefs` defined in the `std` namespace.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### CV Qualifiers in Right Side of using Name

In this example, Polyspace reports violations of this rule when the `const` qualifier is placed on the right hand side of type that uses a `using` name. This violation is not reported for `typedef`-s that are part of the standard (`std`) namespace.

```
#include <cstdint>
#include <string>
using int_p = std::int32_t*;
using int_const_p = int32_t* const;
using const_int_p = const int32_t*;
void Fn(const std::uint8_t& input) // Compliant
{
    std::int32_t value1 = 10;
    std::int32_t value2 = 20;

    const int_p ptr1 =   // Noncompliant deduced type is std::int32_t* const, not const std::int3
    &value1;

    int_p const ptr2 = // Compliant
    &value1;

    int_const_p ptr3 = &value1;    // Compliant

    const_int_p ptr4 = &value1;    //Compliant
    const const_int_p ptr5 = &value1;    // Noncompliant, type is const std::int32_t* const,
                                        //   not const const std::int32_t*
    const_int_p const ptr6 =
    &value1; // Compliant
    const std::string str = "Foo"; //Compliant
}
```

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A7-1-4

The register keyword shall not be used

## Description

### Rule Definition

*The register keyword shall not be used.*

### Rationale

The `register` keyword will be removed from the C++ language in a future release, resulting in code that either no longer compiles or that exhibits undefined behavior. Most compilers ignore the `register` specifier and assign registers themselves. By relying on the presence of specific objects in the processor register, the code might behave in unexpected ways.

To resolve this issue, remove the `register` keyword.

### Polyspace Implementation

Polyspace flags the declaration of an object if its storage type is `register`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Remove `register` Keyword in C++ Source Code

```
#include <cstdint>

int main() {
  register std::int32_t x = 7; //Noncompliant
  std::int32_t y = 7; //Compliant
  return 0;
}
```

In this example, Polyspace flags the use of the keyword `register`.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A7-1-5

The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax

## Description

### Rule Definition

*The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax.*

### Rationale

When you use the `auto` type specifier with a variable declaration, the type of the variable is deduced by the compiler. A developer reading the code might be confused if the type deduction is not what he or she would expect. The information needed to deduce the type might be in a separate part of the code.

This rule permits the use of the `auto` specifier in these cases:

- You declare a variable that is initialized with a function call. This avoids a repetition of the type and ensures that there are no unexpected conversions if you change the return type of the function. For example, in this code snippet, variable `var` has the same type as the return type of `myfunc()`:

  ```
  #include<cstdint>

  int32_t myfunc();

  int foo(){
    auto var=myfunc();
    return var;
  }
  ```
- You declare a variable that you initialize with a non-fundamental type initializer. A type `T` is non-fundamental if `std::is_fundatmental<T>::value` is false. For a list of fundamental types, see Fundamental types. For example, the type of `var` in this code snippet is `std::vector<int>::iterator`.

  ```
  std::vector<int> v = { 1, 2, 3};
  auto var = v.begin();
  ```

  By using the `auto` keyword, you make the code more readable and avoid having to write a difficult to remember non-fundamental type.

  Note that a pointer is a non-fundamental type.
- You declare the parameters of a generic lambda function. The function can then accept more than one kind of parameter types, similar to a function template. For instance, the custom

implementation of `std::sort` in this code snippet can be used to sort vectors of ints, or floats, or other arithmetic types.

```
//sort in ascending order
std::sort(v.begin(), v.end(),
              [](auto lhs, auto rhs){
                  return lhs < rhs});
```

- You use a trailing return type syntax to declare a function template. In this case, there is no type deduction. The `auto` keyword is used as part of an alternative syntax for the declaration of function templates. This code snippet shows an example of trailing return type syntax.

```
template<typename T, typename U>
auto subtract(T lhs, U rhs) -> decltype(lhs - rhs);
```

### Polyspace Implementation

- Polyspace flags the use of the `auto` specifier except when it is used in one of the cases listed in the previous section.
- Polyspace flags the use `auto` to declare a variable that is initialized with a `std::initializer_list` of a fundamental type.
- Polyspace does not flag the use of `decltype(auto)`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of the `auto` Specifier

```
#include <string>
#include<vector>

auto func(int a)   // Non-compliant
{
    return a;
}

void func2()
{
    auto int_type = new int[5];           //  Non-compliant

    auto vector_type = std::vector<int> { 1, 2, 3 }; // Compliant

    const char* c = "hello";
    auto str2 =  std::string(c); // Compliant

    auto lambda = [](auto x, auto y) { // Compliant
        return x > y;
    };


}
```

In this example, the use of the `auto` specifier for the return type of `func` is non-compliant. The deduced return type of the function might not be obvious. Similarly, the use of `auto` in the declaration of `int_type` is non-compliant because the initializer of this variable is an array of type int, which is a fundamental type.

Other uses of `auto` in this example are compliant with the use cases specified by this rule:

- The initializers of `vector_type` and `str2` are non-fundamental types `std::vector<int>` and `stdd::string` respectively (use case (2)).
- The initializer of variable `lambda` is a non-fundametal type lambda expression (use case (2))
- Variables `x` and `y` are parameters of a lambda expression (use case (3)).

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A7-1-6

The typedef specifier shall not be used

## Description

### Rule Definition

*The typedef specifier shall not be used.*

### Rationale

The `using` syntax is a better alternative to `typedef`-s for defining aliases.

Since C++11, the `using` syntax allows you to define template aliases where the template arguments are not bound to a data type. For instance, the following statements define an alias `vectorType` for `vector`, where the argument `T` is not bound to a data type and can be substituted later:

```
template<class T, class Allocator = allocator<T>> class vector;
template<class T> using vectorType = vector<T, My_allocator<T>>;
vectorType<int> primes = {2,3,5,7,11,13,17,19,23,29};
```

The `typedef` keyword does not allow defining such template aliases.

### Polyspace Implementation

The rule checker flags all uses of the `typedef` keyword.

If you do not want to remove certain instances of the `typedef` keyword, add a comment justifying those results. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use of `typedef` Keyword

```
#include <cstdint>
#include <type_traits>

typedef std::int32_t (*fptr1) (std::int32_t); //Noncompliant
using fptr2 = std::int32_t (*) (std::int32_t); //Compliant

template <class T> using fptr3 = std::int32_t (*) (T); //Compliant
```

The alias definitions for `fptr1` and `fptr2` are exactly equivalent. There is no `typedef` equivalent for the alias definition for `fptr3`.

The use of `typedef`-s violates this rule. The rule requires that you stick to the `using` syntax for consistency even when a `typedef` equivalent exists.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A7-1-7

Each expression statement and identifier declaration shall be placed on a separate line

## Description

### Rule Definition

*Each expression statement and identifier declaration shall be placed on a separate line.*

### Polyspace Implementation

The checker raises a violation when two consecutive expression statements are on the same line (unless the statements are part of a macro definition).

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A7-1-8

A non-type specifier shall be placed before a type specifier in a declaration

## Description

### Rule Definition

*A non-type specifier shall be placed before a type specifier in a declaration.*

### Rationale

Non-type specifiers include:

- `typedef`.
- `friend`.
- `constexpr`.
- `register`.
- `static`.
- `extern`.
- `thread-local`.
- `mutable`.
- `inline`.
- `virtual`.
- `explicit`.

To make the code more readable, place non-type specifiers before type specifiers in a declaration.

### Polyspace Implementation

Polyspace flags declarations that place non-type specifiers after a type specifier. If more than one non-type specifiers follow a type specifier, Polyspace flags the rightmost non-type specifier.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Place Non-Type Specifiers Before Type Specifiers

The following example demonstrate the Polyspace implementation of AUTOSAR rule A7-1-8.

```
#include <cstdint>

typedef std::int32_t int1;  // Compliant
std::int32_t typedef int2;  // Noncompliant
```

```
class to_be_friend
{
    explicit to_be_friend(int); // Compliant
    static void* foo(void);     // Compliant
    void static* bar(void);     // Noncompliant
    virtual inline void i1(void) {}; // Compliant
    inline void virtual i2(void) {}; // Noncompliant
    constexpr static long long l1 = 0; // Compliant
    long long constexpr static l3 = 0; //Noncompliant
};
main()
{
    //...
}
```

Polyspace flags declarations where you place non-type specifiers after type-specifiers. The declaration of the static object `l3` is flagged because the non-type specifiers `static` and `constexpr` are placed after the type-specifier `long long`. The violation is highlighted on the rightmost non-type specifier, which is `static`.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A7-1-9

A class, structure, or enumeration shall not be declared in the definition of its type

## Description

### Rule Definition

*A class, structure, or enumeration shall not be declared in the definition of its type.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A7-2-1

An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration

## Description

### Rule Definition

*An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.*

### Rationale

If your program evaluates an expression with `enum` underlying type and returns a value that is not part of the enumerator list of the enumeration, the behavior is unspecified.

Compliance with this rule validates the assumption made by other rules in this standard that objects of type `enum` contain only values corresponding to the enumerators.

To avoid unspecified behavior when evaluating expressions of type `enum`, use a `switch` statement. For example:

```
#include <cstdio>
enum class E : size_t { Pass, Warning, Error };

E Evaluate(size_t input) noexcept {
    E output = E::Pass;
    switch (input) {
      case 0: {
          output = E::Pass;
          break;
      }
      case 1: {
          output = E::Warning;
          break;
      }
      case 2: {
          output = E::Error;
          break;
      }
      default: {
          output = static_cast<E>(0);
      }
    }
    return output;
}
```

### Polyspace Implementation

Polyspace flags the use of a value that is not part of the enumerators of an enumeration.

Polyspace does not flag the use of unknown values, even if you do not check the range of the value before you use it in the `enum` expression.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Value Does Not Correspond to Enumerators of Enumeration

```
#include <cstdint>

enum class Suit : uint8_t { None = 1, Diamonds, Clubs, Hearts, Spades };

Suit resetRedSuit(uint8_t card) noexcept {
    Suit hand = Suit::None;
    if(card < 6){ //check range of card is inside Suit enum
        hand = static_cast<Suit>(card);
    }

    if (hand == Suit::Diamonds ||
        hand == Suit::Hearts) {
            hand = static_cast<Suit>(0); // Noncompliant
    }
    return hand;
}
```

In this example, Polyspace flags the second `static_cast` in function `resetRedSuit` because the expression uses a value (0) that is not part of the enumerator list of enumerator `Suit`.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A7-2-2

Enumeration underlying type shall be explicitly defined

## Description

### Rule Definition

*Enumeration underlying type shall be explicitly defined.*

### Rationale

In an unscoped enumeration declaration such as:

```
enum someEnum : type { ... }
```

if `: type` is omitted, the underlying type is implementation-defined (with the only requirement that the type must accommodate all the enumeration values). Not declaring an underlying type explicitly results in implementation-defined behavior.

In a scoped enumeration declaration such as:

```
enum class someEnum : type { ... }
```

if `: type` is omitted, the underlying type is `int`. If an enumeration value exceeds the values allowed for `int`, you see compilation errors.

For both unscoped and scoped enumerations, declare the underlying type explicitly to avoid implementation-defined behavior or compilation errors.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Enums with Underlying Type Omitted

```
#include <cstdint>

enum E1 { //Noncompliant unscoped enum
    E10,
    E11,
    E12
};

enum E2 : std::uint8_t { //Compliant unscoped enum
    E20,
    E21,
    E22
};

enum class E3 { //Noncompliant scoped enum
```

```
    E30,
    E31,
    E32
};

enum class E4 : std::uint8_t { //Compliant scoped enum
    E40,
    E41,
    E42
};
```

In this example, the code is noncompliant when the underlying types of the enumerations are omitted.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A7-2-3

Enumerations shall be declared as scoped enum classes

## Description

### Rule Definition

*Enumerations shall be declared as scoped enum classes.*

### Rationale

Enumeration values in an unscoped enum can conflict with other identifiers in the same scope as the enum and cause compilation errors. For instance:

```
enum E: std::int32_t { E0, E1};
std::int32_t E0;
```

If you scope the enum, such conflicts can be avoided. For instance:

```
enum class E: std::int32_t { E0, E1};
std::int32_t E0;
```

Scoping the enum also disallows implicit conversions of the enumeration values to other types.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unscoped Enums

```
#include<cstdint>

enum E1: std::int32_t { E10, E11}; //Noncompliant
// std::int32_t E10; causes compilation errors

enum class E2: std::int32_t { E20, E21}; //Compliant
std::int32_t E20;
```

In this example, the declaration of unscoped enum `E1` is noncompliant. Redeclaring an enumeration value of the unscoped enum causes compilation errors (as shown in the commented line that redeclares the enumeration value `E10`).

## Check Information

**Group:** Declaration
**Category:** Required, Automated

## Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A7-2-4

In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized

## Description

### Rule Definition

*In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A7-3-1

All overloads of a function shall be visible from where it is called

## Description

### Rule Definition

*All overloads of a function shall be visible from where it is called.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A7-4-1

The asm declaration shall not be used

## Description

### Rule Definition

*The asm declaration shall not be used.*

### Rationale

The asm declaration is a method to include assembly instructions directly within C++ source code. Support and implementation of the asm declaration is inconsistent across environments. The asm declaration interacts differently with C++ source code in different environments. To avoid restricting the portability of your code, do not use the asm declaration and do not include assembly instructions in your C++ source code.

### Polyspace Implementation

Polyspace flags the use of the asm declaration anywhere in C++ source code.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Assembly Instructions in C++ Source Code

```
#include <cstdint>
using namespace std;
const char* p = "hello world";

void Fn1(void)
{
    asm("movq p, %rdi\n" // Noncompliant
        "call puts");
}

#define _debug() asm volatile("debug":::"memory")  // Noncompliant

void Fn2(void)
{
    _debug();
}

main()
{
    //
}
```

Polyspace flags the use of asm declaration in Fn(1) because the assembly instructions following the declaration are environment-specific. For example, if you use a gcc compiler in a x64 Linux environment, Fn1() produces the string hello world when called. In other environments, the output of the call to Fn1() is unpredictable. Polyspace also flags the use of the asm declaration in creating the _debug() macro.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

Check AUTOSAR C++ 14 (-autosar-cpp14)

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A7-5-1

A function shall not return a reference or a pointer to a parameter that is passed by reference to const

## Description

### Rule Definition

*A function shall not return a reference or a pointer to a parameter that is passed by reference to const.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A7-5-2

Functions shall not call themselves, either directly or indirectly

## Description

### Rule Definition

*Functions shall not call themselves, either directly or indirectly.*

### Rationale

Variables local to a function are stored in the call stack. If a function calls itself directly or indirectly several times, the available stack space can be exceeded, causing serious failure. Unless the recursion is tightly controlled, it is difficult to determine the maximum stack space required.

### Polyspace Implementation

The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.

You can calculate the total number of recursion cycles using the code complexity metric `Number of Recursions`. Note that unlike the checker, the metric also considers implicit calls, for instance, to compiler-generated constructors during object creation.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Direct and Indirect Recursion

```
void foo1(int);
void foo2(int);

void foo1( int n ) {    // Non-compliant - Indirect recursion foo1->foo2->foo1...
    if(n > 0) {
        foo2(n);
        foo1(n);    // Non-compliant - Direct recursion
        n--;
    }
}

void foo2( int n ) { // Non-compliant - Indirect recursion foo2->foo1->foo2...
    foo1(n);
}
```

In this example, the rule is violated because of:

- Direct recursion `foo1` → `foo1`.

- Indirect recursion `foo1` → `foo2` → `foo1`.
- Indirect recursion `foo2` → `foo1` → `foo2`.

Note that the location of the rule violation is different for direct and indirect recursions:

- When a function calls itself directly, the rule violation is shown on the function call.
- When several functions are involved in an indirect recursion chain, for every function in the chain, a rule violation is shown on the function signature in the function body.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A7-6-1

Functions declared with the [[noreturn]] attribute shall not return

## Description

### Rule Definition

*Functions declared with the [[noreturn]] attribute shall not return.*

### Rationale

If you declare a function by using the `[[noreturn]]` attribute, the compiler expects that the function does not return the flow of execution. That is, if a `[[noreturn]]` function `f()` is called from `main()`, then the compiler expects that the flow of execution is not returned to `main()`. If such a function eventually returns the flow of execution, it leads to undefined behavior, which can be exploited to cause data integrity violations.

If a function has no `return` statement, then the final closing brace of a function implies an implicit `return`. Omitting a `return` statement in the function does not prevent the flow of execution from returning. A `[[noreturn]]` function can prohibit returning the flow of execution to the calling function by:

- Entering an infinite loop
- Raising an exception
- Calling another `[[noreturn]]` function

### Polyspace Implementation

If a function specified as `[[noreturn]]` returns the control flow to its caller, Polyspace flags the `[[noreturn]]` function.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Returning The Flow of Execution From `[[noreturn]]` Function

Consider this code containing two `[[noreturn]]` functions.

```
#include <iostream>

[[noreturn]] void noncompliant(int i)
{
    if (i > 0)
    throw "Received positive input";
    else if (i < 0)
    std::exit(0);
} //Noncompliant
```

```
[[noreturn]] void compliant(int i)
{
    if (i > 0)
    throw "Received positive input";
    else if (i < 0)
    std::exit(0);
    else if(i==0)
    while(true){
        //...
    }
}//Compliant
```

- In the `noncompliant()` function, the flow of execution skips the `if-else-if` block of code and returns to the caller implicitly if `i == 0`. Because the `[[noreturn]]` function returns the flow of execution in a code path, this function is noncompliant with this rule.
- In the `compliant()` function:

  - The function raises an exception if `i > 0`.
  - The function calls the `[[noreturn]]` function `std::exit()` if `i < 0`.
  - The function enters an infinite loop if `i==0`.

  Because the `[[noreturn]]` function does not return the flow of execution in any code path, it is compliant with this rule

### Check Information

**Group:** Declaration
**Category:** Required, Automated

## Version History

**Introduced in R2020b**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A8-2-1

When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters

## Description

### Rule Definition

*When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.*

### Rationale

When the return type of a template depends on the types of parameters, using the trailing return type syntax improves readability of the code significantly.

For instance, for out-of-class definitions of methods, using the trailing return type syntax means that you do not have to use the fully qualified return type of a function along with the `typename` keyword. Instead of explicitly specifying the fully qualified return type for `aMethod` in this example:

```
template <typename T>
class aClass {
    public:
      using vectorType = std::vector<T>;
      vectorType aMethod(T const&);
};

//Difficult-to-read method definition
//Part in bold indicates fully qualified return type of method
template <typename T>
typename aClass<T>::vectorType aClass<T>::aMethod(T const &) {
};
```

You can use the trailing return type syntax:

```
template <typename T>
class aClass {
    public:
      using vectorType = std::vector<T>;
      vectorType aMethod(T const&);
};
template <typename T>
auto aClass<T>::aMethod(T const &) -> vectorType {
};
```

### Polyspace Implementation

The checker flags function template declarations where the explicitly specified return type of a template function has the same scope as the template function itself.

For instance, in the preceding example, the function `aMethod` has a return type `vectorType`, which has the same scope as `aMethod`, namely the class `aClass<T>`. Instead of explicitly specifying the fully qualified return type, you can use the trailing return type syntax.

Because C++14 has enabled return-type deduction, you can use the `auto` keyword to declare generic templates while omitting the trailing return type. In such cases, Polyspace does not raise a violation.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Explicitly Specifying Return-Types of Generic Templates

```
#include <vector>
#include<cstdint>

template<class T, class U>
decltype(std::declval<T>() * std::declval<U>())
bar(T const& lhs, U const& rhs) {// Noncompliant
  return lhs * rhs;
}

template<class T, class U>
auto foo(T a, U b) -> decltype(a*b){        //Compliant
  return a*b;
}
template<class T, class U>
auto foo2(T a, U b) {        //Compliant
  return a*b;
}
```

In this example, three generic function templates are declared:

- The template `bar` explicitly defines the return-type. Such declarations are difficult to read and understand. Polyspace flags the declaration.
- The template `foo` uses the keyword `auto`, and then specifies a trailing return-type. Such declarations are easy to read and understand. Polyspace does not flag the declaration.
- The template `foo2` uses the keyword `auto` but omits the trailing return-type. From C++14 onward, the compiler can deduce the return type of such templates. Polyspace does not flag the declaration.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**
"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A8-4-1

Functions shall not be defined using the ellipsis notation

## Description

### Rule Definition

*Functions shall not be defined using the ellipsis notation.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A8-4-2

All exit paths from a function with non-void return type shall have an explicit return statement with an expression

## Description

### Rule Definition

*All exit paths from a function with non-void return type shall have an explicit return statement with an expression.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A8-4-3

Common ways of passing parameters should be used.

## Description

### Rule Definition

*Common ways of passing parameters should be used.*

### Rationale

If you follow well-established conventions for passing parameters, a developer or reviewer can determine from your function signature whether a parameter is an input parameter, an output parameter, or a different type of parameter.

These conventions are commonly used to pass parameters to a function `f(X)`:

- **In**: If the input parameter data type `X` is cheap to copy or cannot be copied (for instance, the `std::unique_ptr` type), pass the parameter by value as `f(X)`. Otherwise, pass the parameter by `const` reference as `f(const X &)`.
- **Out**: If the output parameter data type `X` is expensive to move, pass the parameter by reference as `f(X &)`. Otherwise, do not pass a parameter, but instead return the value as `X f()`.
- **In/Out**: Pass the parameter by lvalue reference as `f(X &)`.
- **Consume**: Pass the parameter by rvalue reference as `f(X &&)`.
- **Forward**: Pass the parameter by template rvalue reference as `template<typename T> f(T &&)`.

### Polyspace Implementation

The checker flags these incorrect ways of passing parameters:

- **In** parameters:

  - You pass by value an input parameter that is expensive to copy:

    ```
    f(X); //X is expensive to copy
    ```
  - You pass by reference an input parameter that is cheap to copy:

    ```
    f(X &); //X is cheap to copy
    f(const X &); //X is cheap to copy
    ```

  The checker considers a data type that has a size less than twice `sizeof(void *)` as cheap to copy.
- **Out** parameters:

  - You return by value an output parameter that is expensive to move:

    ```
    X f(); //X is expensive to move
    ```
  - You pass by reference an output parameter that is cheap to move:

    ```
    f (X &); //X is cheap to move
    ```

The checker considers a fundamental data type that has a size less than eight times `sizeof(void *)` as cheap to move.

The checker does not include pass by pointers under the umbrella of pass by reference.

Resolving a violation of this rule can sometimes involve invasive changes. For instance, if the checker suggests that an output parameter can be returned by value, but the function already returns a value, you have to rewrite your code significantly. You have to combine the output parameter and already returned value into a structure or n-tuple and then return this structure or n-tuple. If the output parameter and already returned value are not semantically related, combining them into a structure might not be appropriate. In this case, add a comment to the result or code to avoid another review. See "Address Results in Polyspace User Interface Through Bug Fixes or Justifications".

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Input Parameters

```
#include <iostream>
#include <array>
#include <numeric>

typedef struct smallStruct {
    char x;
    char y;
}smallStruct;

void sum(smallStruct& aStruct) { //Noncompliant
    std::cout<<aStruct.x+aStruct.y;
}



typedef struct largeStruct {
    std::array<int,20> arrayOfIntegers;
    int init;
}largeStruct;

void add(largeStruct aStruct) { //Noncompliant
    std::cout<<std::accumulate(aStruct.arrayOfIntegers.begin(),
                     aStruct.arrayOfIntegers.end(), aStruct.init);
}
```

In this example, functions `sum` and `add` only read the contents of their parameters. Therefore, these parameters are input parameters.

- The function `sum()` with signature:

  ```
  void sum(smallStruct& aStruct);
  ```

  takes as argument an object of type `smallStruct` that is cheap to copy. The argument can be passed by value instead of reference.

- The function `add()` with signature:

  ```
  void add(largeStruct aStruct);
  ```

  takes as argument an object of type `largeStruct` that is expensive to copy. The argument can be passed by reference instead of value.

### Output Parameters

```
#include <array>
#include <algorithm>

void init(int& val) { //Noncompliant
    val = 0;
}

typedef struct largeStruct {
    std::array<int,100> arrayOfIntegers;
    int init;
}largeStruct;

largeStruct reset(void) { //Noncompliant
    largeStruct aStruct;
    std::fill(aStruct.arrayOfIntegers.begin(), aStruct.arrayOfIntegers.end(), 0);
    aStruct.init = 0;
    return aStruct;
}
```

In this example, functions `init` and `reset` only write to the contents of their parameters. Therefore, these parameters are output parameters.

- The function `init()` with signature:

  ```
  void init(int& val);
  ```

  takes as an argument an object of type `int` that is cheap to move. The object can be returned by value instead of being passed by reference.
- The function `reset()` with signature:

  ```
  largeStruct reset(void);
  ```

  takes as an argument an object of type `largeStruct` that is expensive to move. The object can be passed by reference instead of being returned by value.

## Check Information

**Group:** Declarators
**Category:** Advisory, Non-automated

# Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A8-4-4

Multiple output values from a function should be returned as a struct or tuple

## Description

### Rule Definition

*Multiple output values from a function should be returned as a struct or tuple.*

### Rationale

In a C++ function, the `return` statement can return only the value stored in a single variable. But the values stored in any number of additional variables in the caller's scope can be modified by the callee if you pass these values by reference, and then modify them in the body of the callee. For example, consider the function `foo`:

```
int foo(int x, int& y)
{
  int z;
  y = x*x;
  z = x*x*x;
  return z;
}
```

The function `foo` effectively returns two integer values: the square of the input parameter x (returned by reference) and the cube of the input parameter x (returned by copy by using the `return` statement). Simultaneously using both strategies to return multiple values results in a complicated function interface and can make your code less readable and maintainable. Instead, storing all return values in a single struct or tuple and returning it by using the `return` statement results in a simpler, more unified interface.

A `return` statement that has a struct or a tuple might require expensive copying from one memory location to another. Most compilers support return value optimization and can eliminate this expensive copy, resulting in executable code with little to no overhead associated with such returns.

To help you decide whether to use a struct or a tuple to return multiple values, consider:

- If your return type represents an abstraction, it is preferable to use a struct because you can provide a custom name for each component of the abstract data type.
- Tuples are easier to work with because a returned tuple can be conveniently processed by using `std::tie` at the call site. The `std::tie` method puts the tuple elements directly into existing local variables in the caller.

**Note** This rule also applies to `std::pair`, which is a special kind of tuple that has exactly two elements.

### Polyspace Implementation

The checker flags a function declaration that satisfies one of these two conditions:

- The function has a nonvoid return type and at least one nonconstant reference parameter
- The function has more than one nonconstant reference parameters

Usage notes and limitations:

- The checker flags pure virtual functions that violate this rule. These functions are flagged because, for any implementation of a pure virtual function to be compliant with this rule, the interface of the pure virtual function itself must obey this rule.
- The checker does not flag operators that violate this rule.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Combine Multiple Return Values into a Tuple

```
#include <tuple>


int Divide1(int dividend, // Noncompliant, remainder returned as reference parameter
            int divisor, int& remainder)
{
  remainder = dividend % divisor;
  return dividend / divisor;
}

// Compliant, quotient and remainder combined into a tuple
std::tuple<int, int> Divide2(int dividend, int divisor)
{
  return std::make_tuple(dividend / divisor, dividend % divisor);
}

int main()
{
  int quotient, remainder;
  // store in local variables
  std::tie(quotient, remainder) = Divide2(26, 5);
  return 0;
}
```

The function `Divide1` has the quotient as the return value and the remainder as a nonconstant reference parameter. Having a nonvoid return value and a nonconstant reference parameter violates this coding rule.

The function `Divide2` combines the quotient and the remainder into a tuple and returns the tuple. This code pattern complies with the rule.

### Combine Multiple Return Values into a Structure

```
struct fraction {
  int quotient;
  int remainder;
} ;



int Divide1(int dividend, // Noncompliant, quotient and remainder returned as reference parameters
            int divisor, int& quotient, int& remainder)
{
  quotient = dividend / divisor;
  remainder = dividend % divisor;
}

// Compliant, quotient and remainder combined into a struct
```

```
fraction Divide2(int dividend, int divisor)
{
  fraction answer;
  answer.quotient = dividend / divisor;
  answer.remainder = dividend % divisor;
  return answer;
}

int main()
{
  fraction answer;
  answer = Divide2(26,5);
  return 0;
}
```

The function `Divide1` has both the quotient and the remainder as nonconstant reference parameters. Having multiple nonconstant reference parameters violates this coding rule

The function `Divide2` combines the quotient and the remainder into a struct and returns the struct. This code pattern complies with the rule.

## Check Information

**Group:** Declarators
**Category:** Advisory, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A8-4-5

"consume" parameters declared as X && shall always be moved from

## Description

### Rule Definition

*"consume" parameters declared as X && shall always be moved from.*

### Rationale

When declaring a function, you might indicate your intention of moving the content of a function parameter by declaring it as a nonconst and nontemplate rvalue reference or a "consume" (X&&) parameter. For instance, the parameter of this function is declared as a "consume" parameter: `void foo(std::vector<std::string>&& V)`. This declaration implies that the content of the vector `V` is intended to be moved instead of copied within the body of the function.

When you declare a function parameter as a "consume" parameter, use move semantics when using the parameter. Within the body of the function, use the `std::move` function explicitly if you use an lvalue reference to invoke the function.

### Polyspace Implementation

Polyspace flags the definition of a function if both of these conditions are true:

- At least one function parameter is declared as a nonconst and nontemplate rvalue reference, that is, a "consume" or X&& parameter.
- The content of the X&& parameter is not completely moved to another object by using the `std::move` function within the body of the function.

Polyspace does not raise this defect in move constructors and move assignment operators.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use `std::move` on Nonconst and Nontemplate rvalue Reference Function Parameters

```
#include <utility>
class C
{
    C(C&& c): a{std::move(c.a)}     // Compliant by exception.
    {
    }

    C& operator=(C&& c)             // Compliant by exception.
    {
        a = std::move(c.a);
        return *this;
    }

    void move(C&& c)                // Noncompliant
    {
        a = std::move(c.a);//Partial move
    }

    void cond(C&& c, bool b)        // Compliant
    {
        if (b) {
            move(std::move(c));
        } else {
            a++;
        }
    }
public:
    int a;
    void set(int&& num)     // Compliant
    {
        a = std::move(num);
    }
    void set1(int&&)     // Noncompliant
    {
        //Unnamed temporary variable cannot be moved from.
    }
    void set2(int&& i12);  // Violation raised on definition.
    void set3(int&& i11a,  // Noncompliant
    int&& i11b)  // Noncompliant
    {
        if(i11a != i11b)
        {
        }
    }

};
void C::set2(int&& i12)   // Noncompliant
{
    a = i12;
}

template<typename T>
void tf1(T&& t1)      // Compliant - not a "consume" parameter
{
}
```

In this example, the data member `a` of the class `C` is set to an integer by using move semantics.

- Polyspace does not flag the move constructor and the move assignment operator even though these functions do not completely move the "consume" or X&& parameter. These functions are compliant by exception.
- Polyspace flags the function `C::move` because the body of the function partially moves the "consume" or X&& parameter.
- Polyspace flags the function `C::set1` because this function uses an unnamed "consume" parameter. Because the parameter is unnamed, you cannot use the function `std::move` on this X&& parameter.

- Polyspace flags the function `C:: set2` because the body of the function copies the "consume" parameter instead of using the function `std::move`. Polyspace raises the violation on the definition of the variable. Similarly, the function `C::set2` is also noncompliant with this rule because its body does not use `std::move` on X&& variables.
- The function `C::cond` and `C::set` are compliant with this rule because the bodies of these functions use `std::move` on the "consume" parameters.
- Polyspace does not flag the function template because this rule does not apply to templates.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A8-4-6

"forward" parameters declared as T && shall always be forwarded

## Description

### Rule Definition

*"forward" parameters declared as T && shall always be forwarded.*

### Rationale

Because rvalue references cannot bind to lvalues, functions that enable the use of move semantics by using rvalue references in their signature do not accept lvalues. This issue is resolved by using an rvalue reference to a nonconst template type object, which is called a "forward" parameter. These parameters can bind to both rvalues and lvalues while preserving their cv qualifications and value categories. "Forward" parameters are useful when you want to forward a value to a destination object or function by using the function `std::forward`.

When you declare a function template by using a "Forward" parameter, do not use the parameter in any operations. Because "Forward" parameters can bind to both lvalues and rvalues, using them in an operation might corrupt their cv qualifications and value categories. Forward these parameters directly to the destination by using `std::forward` without using them in an operation.

### Polyspace Implementation

Polyspace flags a "Forward" parameter in the definition of a function template or a Lambda expression if any of these conditions are true:

- A "Forward" parameter is not forwarded to a destination by using `std::forward`.
- An operation other than forwarding is performed on the "Forward" parameter or on a member object of it.

Polyspace ignores the templates and Lambda expressions that remain unused in your code.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Compliant Use of "Forward" Parameters

```
#include<string>
#include<vector>
#include <iostream>

struct intWrapper {
    intWrapper(int&& n) { std::cout << "rvalue overload, n=" << n << "\n"; }
    intWrapper(int& n)  { std::cout << "lvalue overload, n=" << n << "\n"; }
};
```

```
struct floatWrapper {
    floatWrapper(double&& n) { std::cout << "rvalue overload, n=" << n << "\n"; }
    floatWrapper(double& n)  { std::cout << "lvalue overload, n=" << n << "\n"; }
};

class mixedNumerical {
public:
    template<class T1, class T2, class T3>
    mixedNumerical(T1&& t1, T2&& t2, T3&& t3) :   //violation on T3
    a1_{std::forward<T1>(t1)},
    a2_{std::forward<T2>(t2)},
    a3_{std::forward<T3>(t3)}
    {
    }

private:
    intWrapper a1_, a2_;
    floatWrapper    a3_;
};


template<class T, class... U>
std::unique_ptr<T> unique_ptr_factory(U&&... u)
{
    return std::unique_ptr<T>(new T(std::forward<U>(u)...));
}

int main()
{
    auto p1 = unique_ptr_factory<intWrapper>(2); // rvalue
    int i = 1;
    auto p2 = unique_ptr_factory<intWrapper>(i); // lvalue

    std::cout << "mixedNumerical\n";
    double lvalue = 2.2;
    auto t = unique_ptr_factory<mixedNumerical>(2, i, 2.2);// rvalue
    auto t2 = unique_ptr_factory<mixedNumerical>(2, i, lvalue);// lvalue
}
```

This example shows the implementation of a flexible interface to the function `unique_ptr_factory` by using a "forward" parameter pack. This function accepts the "forward" parameter pack, and then forwards the parameters to their respective constructors by using `std::forward`. The constructors are overloaded to accept both rvalues and lvalues. As a result, the function `unique_ptr_factory` produces `unique_ptr` to `intWrapper` type objects and `mixedNumerical` type objects while requiring minimal overloading. This use of "forward" is compliant with this rule because the "forward" parameters are forwarded to their destination by using `std::forward`. Because no other operation is performed on them, their cv qualification and value categories are preserved.

### Noncompliant Use of "Forward" Parameters

```
#include<string>
#include<vector>
void task(int i);
template<typename T>
T NoncompliantTemplate(T&& arg)    // Noncompliant
```

```
{
    return arg;       // Noncompliant
}

auto NoncompliantLambda = [](auto&& truc) {    // Noncompliant
    return truc;                           // Noncompliant
};
template<typename T>
T ReturnStaticCast(int&& i) //Compliant: not a template parameter.
{
    return static_cast<T>(i);
}
template<typename T>
void ConstArg(const T&& t)      // Compliant: const
{}

template<typename T>
void UnusedArg(T&& t)              // Noncompliant
{}

template<typename T>
void UnnamedArg(T&& )              // Noncompliant
{}
template<typename T>
void usage(T&& t1, T&& t2)
{
    if (t1==t2)                       // Noncompliant
    {
        task(std::forward<T>(t1));
    }
    else
    {
        task(std::forward<T>(t1));
        task(std::forward<T>(t2));
    }
}
class intWrapper
{
public:
    int m;
};

template<typename T>
void CheckForward(T&& t)
{
    if (t.m != 0)                          // Noncompliant
    {
        UnusedArg(std::forward<T>(t));
    }
}
auto CompliantLambda = [](auto&& truc) {    // Compliant
    return NoncompliantLambda(std::forward<decltype(truc)>(truc));
};
template<typename T>
T NoninstantiatedTemplate(T&& arg)    // Not checked
{
    return arg;       // Not checked
}
```

```
void foo(){
    int i;
    intWrapper C;
    C.m = i;
    NoncompliantTemplate(i);
    CheckForward(std::move(C));
    usage(i,C.m);
    UnnamedArg(i);
    CompliantLambda(i);
}
```

This example shows use of "forward" parameters that are not compliant with this rule.

- Polyspace flags the nonconst `T&&` parameter `arg` of the template `NoncompliantTemplate` because this "forward" parameter is not forwarded to a destination by using `std::forward`. The parameter is flagged in the declaration and the return statement.
- Polyspace flags the nonconst `auto&&` parameter `truc` of the Lambda expression `NoncompliantLambda` because this "forward" parameter is not forwarded to a destination by using `std::forward`. The parameter is flagged in the declaration and the return statement.
- Polyspace does not flag `int&&` argument `i` of the template `ReturnStaticCast` because this argument is not a nonconst template type rvalue reference. For the same reason, Polyspace does not flag the argument of `ConstArg`.
- Polyspace flags the nonconst template type rvalue reference argument `t` of the template `UnusedArg` because this "forward" parameter is not forwarded to a destination by using `std::forward`.
- Polyspace flags the argument of the template `UnnamedArg` because the "forward" parameter is unnamed and it cannot be forwarded by using `std::forward`.
- Polyspace flags the parameters `t1` and `t2` in the statement `if (t1==t2)` in the template `usage` because these "forward" parameters are used in an operation before they are forwarded by using `std::forward`. This checker is also raised on `t` in the statement `if (t.m != 0)` in the template `CheckForward` because a member of the "forward" parameter `t` is accessed.
- Polyspace does not check the template `NoninstantiatedTemplate` because this template is unused in the code.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A8-4-7

"in" parameters for "cheap to copy" types shall be passed by value

## Description

### Rule Definition

*"in" parameters for "cheap to copy" types shall be passed by value.*

### Rationale

You use an "in" parameter when you intend to only read that parameter within a function. If the parameter is cheap to copy, pass the parameter by value to:

- Make it clear that you do not plan on modifying the parameter.
- Avoid the additional indirection that is required to access the parameter from the function when you pass the parameter by reference.

A parameter is cheap to copy when both these conditions are true:

- The parameter has a size less than or equal to two words. For instance, for a parameter `foo`, `sizeof(foo) <= 2 * sizeof(int)`.
- The parameter is trivially copyable type. See *is_trivially_copyable*.

### Polyspace Implementation

- Polyspace flags:
  - `const` parameters that are passed by reference if the parameters are cheap to copy (`sizeof <= 2 * sizeof(int)` and trivially copyable).
  - `const` parameters that are passed by value if the parameters are not cheap to copy. For instance, in this code snippet, both parameters `str` (expensive to copy) and `b` (non-trivially copyable) are noncompliant.

    ```
    void func1(const std::string str);
    struct B {
        B(B const&) {}
    };
    void func2(const B b);
    ```

    .
- Polyspace does not flag :
  - Non-`const` parameters that are passed by reference if those parameters are not cheap to copy and are not modified inside the function. Polyspace considers these parameters as "in" parameters.
  - "in" parameters that are passed by reference if those parameters are move-only types. For instance, `int f(const std::unique_ptr<int>& p);`.
  - `const` parameters that are passed by reference in copy constructors. For instance, no defect is raised on `point` in this code snippet.

```
class coord
{
public:
    coord(int x, int y) {p_x = x; p_y = y;}
    coord(const coord& point) { p_x = obj.p_x; p_y = obj.p_y;}
    //...
private:
    int p_x, p_y;


};
coord point{1, 1};
void func(const coord& point);
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### "in" Parameters Passed by Reference

```
#include <memory>
#include <string>
#include <cstdint>

int func(const std::unique_ptr<int>& ptr) // Compliant
{
    *ptr = *ptr + 1;
    return *ptr;
}
union Small {
    uint8_t var1 ;
    uint8_t var2;
} ;

struct Large {
    std::uint32_t v1;
    std::uint32_t v2;
    std::uint32_t v3;
    std::uint32_t v4;
};

void func2(Small& arg) // Noncompliant
{
//...
}

void func3(Large val,   // Noncompliant
           std::string& str) // Compliant
{
//...
}
```

In this example, Polyspace flags "in" parameters:

- `arg`, because it is passed by reference and its type is trivially copyable. This parameter can be passed by value instead.
- `val`, because it is passed by value and it is expensive to copy. Passing this parameter by reference avoids making expensive copies for each call to `func3()`.

These passed by reference "in" parameters are compliant:

- Parameter `ptr` because it is a move-only type.
- Parameter `str` because it is expensive to copy. This parameter is non-`const` but it is not modified inside `func3()`.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A8-4-8

Output parameters shall not be used

## Description

### Rule Definition

*Output parameters shall not be used.*

### Rationale

You can store the output value of a function in a variable that you pass to that function as a non-const reference or pointer parameter, for example:

```
void func(const T* input_var, T* output_var); //declaration
void func(const T* input_var, T* output_var)
{
    *output_var = *input_var % 2;
}
```

However, it is unclear from the function declaration whether the output parameter `output_var` passes a value to `func` and then stores the output (in-out parameter), or whether `output_var` only stores the output (out parameter). This might cause a developer to misuse the parameter, for instance by passing a null parameter when the function expects a non-null parameter.

Instead, use a return value to store the function output. The return value makes your intent clear and prevents possible misuse of the passed parameters, for example:

```
T* func(const T* input_var)
{
    return *input_var % 2;
}
```

### Polyspace Implementation

Polyspace flags all uses of non-const references or pointers in the parameter list of:

- Functions, except for `main()`.
- Class constructors and operators.

If your code contains a function declaration and its definition, Polyspace flags the violation in the function definition.

**Note** Polyspace flags a non-const reference or pointer in parameter lists even if that parameter is not used as an output parameter.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Possible Output Parameters

```
#include<iostream>

std::int32_t func(const std::vector<int32_t>& inParam,
                  std::vector<int32_t>& outParam)  //  Non-Compliant
{
    //...
    return 1;
}

class C
{
public:

    C(C* ptr) {}         //  Non-Compliant
    C(C& ref) {}         //  Non-Compliant
    C(C&& rvalue_ref) {} //  Compliant
    C(const C& c) {}     //  Compliant
    C(const C&& c) {}    //  Compliant

    C& operator=(C& ref) { return *this; }    //    Non-Compliant

};
```

In this example, `func` has a return value of type `std::int32_t` but its parameter list is still non-compliant because it contains non-const lvalue reference `outParam`. Similarly, non-const parameters `ptr` and `ref` in the class constructors and in `operator=` are non-compliant.

Note that non-const parameter `rvalue_ref` is compliant because rvalue reference parameters bind only to temporary objects and these objects cannot be referenced after the function goes out of scope.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A8-4-9

"in-out" parameters declared as T & shall be modified

## Description

### Rule Definition

*"in-out" parameters declared as T & shall be modified.*

### Rationale

A function parameter meant to be both read and modified within a function is called an "in-out" parameter.

If you do not both read and modify a parameter, avoid passing by non-`const` reference so that the function prototype reflects the true nature of the parameter.

- If you only read a parameter within a function, the parameter is actually an "in" parameter.

  Pass the parameter by `const` reference.
- If you replace the entire contents of a parameter within a function, the parameter is actually an "out" parameter.

  If possible, avoid "out" parameters completely and store any output of the function in the function return value. See also `AUTOSAR C++14 Rule A8-4-8`.

### Polyspace Implementation

The checker checks each function parameter passed by non-`const` reference and raises a violation if the parameter is only read within the function or its value completely replaced within the function.

The checker does not raise a violation if:

- The parameter is an object and you access one or more of its data members, or invoke a non-`const` member function.
- You pass a pointer or reference to the parameter on to another function.
- The function is virtual. The reason is that even if the current function might not modify its parameter, an override of the function might modify its corresponding parameter.
- The function is an unused class method.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### "In-out" Parameter Not Modified in Function Body

```
#include <cstdint>
#include <vector>
```

```
#include <numeric>
#include <string>

int32_t addAllElements (std::vector<int32_t>& aVec) { //Noncompliant
     return std::accumulate(aVec.cbegin(), aVec.cend(), 0);
}

int32_t addEveryElement (const std::vector<int32_t>& anotherVec) { //Compliant
     return std::accumulate(anotherVec.cbegin(), anotherVec.cend(), 0);
}
```

In this example, the vector `aVec` is passed as a non-`const` reference to the function `addAllElements`. However, the vector is only read within the function and is only an "in" parameter, not an "in-out" parameter.

The function `addEveryElement` is a compliant version of the same function. The "in" parameter `anotherVec` is passed as a `const` reference.

**"In-out" Parameter Fully Replaced in Function Body**

```
#include <string>

void replaceString(std::string &Source, const std::string Replacement) { //Noncompliant
    if(Replacement.at(0)=='_')
         Source = Replacement;
    else
         Source = "_null";
}

std::string replacementString(const std::string str) { //Compliant
    if(str.at(0)=='_')
         return str;
    else
         return "_null";
}
```

In this example, the string `Source` is passed as a non-`const` reference to the function `replaceString`. However, the string is fully replaced within the function and is only an "out" parameter, not an "in-out" parameter.

The function `replacementString` is a compliant version of the same function, which also does not violate `AUTOSAR C++14 Rule A8-4-8`. The function has the same output as `replaceString` but stores the output in its return value.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A8-4-10

A parameter shall be passed by reference if it can't be NULL

## Description

### Rule Definition

*A parameter shall be passed by reference if it can't be NULL*

### Rationale

A reference cannot be NULL. If a parameter is required (it cannot be NULL), pass that parameter by reference to make your intent clearer. Passing by reference also yields cleaner code because you do not need to check whether the parameter is NULL before you use it.

### Polyspace Implementation

Polyspace flags passed-by-pointer parameters except if:

- The pointer is a smart pointer such as `std::shared_ptr`.
- The pointer is not dereferenced in the function.
- The pointer is checked against NULL, even if the check happens after the dereference.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Pass-by-Pointer Parameters

```
#include <iostream>
#include <vector>

void other_func(std::vector<int32_t>*);

void func(std::vector<int32_t>* v_ptr1, //  Non-Compliant
             std::vector<int32_t>* v_ptr2, //  Compliant
             std::vector<int32_t>* v_ptr3) //  Compliant
{

    auto v = v_ptr1;
    auto ptr_size = 0;
    if (v_ptr2 != NULL) {
        ptr_size = v_ptr2->size();
    }
    v->resize(ptr_size);

    other_func(v_ptr3);
    //....
}
```

In this example pass-by-pointer parameter `v_ptr1` is non-compliant because it is dereferenced inside `func` without checking if the pointer is NULL. If `v_ptr1` cannot be NULL, pass this parameter by reference. If the `v_ptr1` can be NULL, check whether the pointer is NULL before you dereference it to avoid a segmentation fault.

Parameter `v_ptr2` is compliant because it is checked against NULL, which indicates it could be NULL.

`v_ptr3` is compliant because it is not dereferenced inside `func`.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A8-4-11

A smart pointer shall only be used as a parameter type if it expresses lifetime semantics

## Description

### Rule Definition

*A smart pointer shall only be used as a parameter type if it expresses lifetime semantics.*

### Rationale

A smart pointer manages the lifetime of a dynamically allocated object and destroys that object when the smart pointer goes out of scope. When you use a smart pointer as a parameter type for a function, you restrict that function to callers that use smart pointers, which is unnecessary if the function does not affect the lifetime of the managed object. In the case of shared smart pointers such as `std::shared_ptr`, your program incurs additional run-time overhead.

Examples of functions that affect the lifetime of a managed object include:

- A function that transfers ownership of the managed object from the `std::unique_ptr` smart pointer parameter to a different `std::unique_ptr` smart pointer through a move operation.
- A function that copies the `std::shared_ptr` smart pointer parameter inside the function body, sharing the ownership of the managed object with another `std::shared_ptr` smart pointer.

If the function does not affect the lifetime of the managed object, pass the object by reference or as a raw pointer instead.

If that object is managed by a non-local smart pointer, check that both these conditions are true:

- The object is not reset or reassigned inside a function that is down the call tree of the called function.
- The managed object is not destroyed before it is dereferenced.

For instance, if the object is managed by a non-local shared pointer, make a local copy inside the called function.

The use of a non-owning smart pointer as a parameter type, such as `std::observer_ptr`, is compliant with this rule. See `std::experimental::observer_ptr`.

### Polyspace Implementation

Polyspace reports a violation of this rule when you use smart pointers `std::shared_ptr` or `std::unique_ptr` as parameter types in a user-defined function that does not affect the lifetime of those smart pointers in the function body.

Polyspace considers the lifetime of an `std::shared_ptr` smart pointer parameter affected if, along at least one execution path:

- The pointer is copied through a copy constructor or a copy assignment.
- The pointer is moved-from through a move constructor or move assignment. This operation involves a call to `std::move`.

- The pointer is passed as an argument to a function with one of these parameter types:
  - An `std::shared_ptr<T>` type. In this case the copy constructor is invoked.
  - A `const std::shared<T>&` type. This `const` lvalue reference indicates that the function retains a reference count and might make a copy of the pointer.
  - An `std::shared<T>&` type. This lvalue reference indicates that the passed parameter will be modified.
- A modifier member function such as `std:shared_ptr::reset` or a modifier non-member function such as `std::shared_ptr::swap` is called on the pointer. For a list of modifier functions, see std::shared_ptr.
- The pointer is the destination of a copy or move assignment.

`const`-qualified `std::shared_ptr` smart pointer parameters cannot be modified.

Polyspace considers the lifetime of an `std::unique_ptr` smart pointer parameter affected if, along at least one execution path:

- The pointer is moved-from with a call to `std::move`.
- The pointer is moved-from through a move constructor or a move assignment.
- The pointer is passed as an argument to a function with one of these parameter types:
  - An `std::unique_ptr<T>` type. In this case, the move constructor is invoked.
  - An `std::unique<T>&` type. This lvalue reference indicates that the function modifies the passed parameter.
- A modifier member function such as `std::unique_ptr::reset` or a modifier non-member function such as `std::unique_ptr::swap` is called on the pointer. For a list of modifier functions, see std::unique_ptr.
- The pointer is the destination of a move assignment.

`const`-qualified `std::unique_ptr` smart pointer parameters cannot be moved-from or modified and their lifetimes are not affected. The lifetime of the passed argument is also not affected in the case of the `const` lvalue reference parameter `const std::unique_ptr<T>&`.

Polyspace does not report a violation of this rule when you use smart pointers as parameter types in these cases:

- The pointer is used as a parameter of a template function. The parameter types of a template function depend on the instantiation of the function and a fix is not always possible at the template design level. For instance, the use of the shared pointer `ptr` is not reported as a violation in this code.

  ```
  #include <iostream>
  #include <memory>
  #include <cassert>

  template <typename T>
  double XtimesY(T ptr) // Use of ptr not flagged
  {
    return (ptr->x) * (ptr->y);
    // ...
  }
  struct S
  ```

```
{
  double x;
  double y;
  S(double x0, double y0) : x{x0}, y{y0} {}
};
void func()
{
  auto a = new S{100.0, 200.0};
  auto a_cpy = std::shared_ptr<S>(a);
  assert(XtimesY(a) == XtimesY(a_cpy));
}
```

- The smart pointer parameter is captured by a `lambda` function inside the function body. For instance, Polyspace does not report a violation when you use the shared pointer `ptr` as a parameter of `func()` in this code. The parameter is captured by a `lambda` function inside the body of `func()`. Polyspace reports the use of `lambda` function parameters `sp1` and `up1` as violations because their lifetime is not modified inside the `lambda` function.

```
#include <memory>

struct S
{
  double x;
  double y;
  S(double x0, double y0) : x{x0}, y{y0} {}
};

void func(std::unique_ptr<S> ptr)
{ // no defect is detected on 'ptr'
  auto lambdaF =
      [&ptr0 = ptr]            // 'ptr' is captured (by reference)
      (std::shared_ptr<S> sp1, // Non-compliant
       std::unique_ptr<S> up1) // Non-compliant
  {                            // lifetime of 'sp1' and 'up1' not affected
    // ...
  };
  auto a = std::make_shared<S>(1.0, 2.0);
  auto b = std::make_unique<S>(100.0, 200.0);
  lambdaF(a, std::move(b));
}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### shared_ptr Used as Function Parameter Type

```
#include <memory>
#include <cassert>
#include <iostream>
#include <string>

class A
{
};

//

void ptr_is_moved_from(
    std::shared_ptr<A> ptr,       // Compliant
    const std::shared_ptr<A> ptr1 // Compliant
)
{

  std::shared_ptr<A> destinationPtr = std::move(ptr);
  destinationPtr = std::move(ptr1); // copy takes place
}
```

```
//


void ptr_param_is_lvalue_ref(std::shared_ptr<A> &ptr)
{
  ptr.reset(new A()); // ptr is modified by member function
}
void pass_to_function_with_lvalue_ref_param(
    std::shared_ptr<A> ptr,       // Compliant
    const std::shared_ptr<A> ptr1 // Non-compliant
)
{

  ptr_param_is_lvalue_ref(ptr);
#if 0
        // compilation error
        use_lvalue_reference_shared_ptr(ptr1); //ptr1 is const-qualified and cannot be modified
#endif
}

//

void member_function_modifies_ptr(
    std::shared_ptr<A> ptr,       // Comliant
    const std::shared_ptr<A> ptr1 // Non-compliant
)
{
  ptr.reset();
#if 0
        // compilation error
        ptr1.reset(); //ptr1 is const-qualified and cannot be modified

#endif
}

//

void ptr_is_destination_of_copy_move(
    std::shared_ptr<A> ptr,       // Comliant
    const std::shared_ptr<A> ptr1 // Non-compliant
)
{

  std::shared_ptr<A> gPtr0 = std::make_shared<A>();
  ptr = gPtr0; // ptr destination of copy assignment
#if 0
        // compilation error
        ptr1 = gPtr0; // ptr1 is const-qualified and cannot be modified
#endif
}
```

In this example, Polyspace reports the use of the shared pointer parameter `const std::shared_ptr<A> ptr1` as a violation because the parameter is `const`-qualified and cannot be modified. The lifetime of the parameter is not affected inside these functions and running the program results in a compilation error.

- Function `pass_to_function_with_lvalue_ref_param()`
- Function `member_function_modifies_ptr()`
- Function `ptr_is_destination_of_copy_move()`

Polyspace does not report the use of `const std::shared_ptr<A> ptr1` in the function `ptr_is_moved_from()` because the call to `std::move` creates a copy, which modifies the lifetime of `ptr1` by incrementing the reference count.

Polyspace does not report the use of the shared pointer parameter `std::shared_ptr<A> ptr` in these functions because, in each case, the lifetime of the smart pointer is affected:

- In the function `ptr_is_moved_from()`, the managed object is moved-from.

- In the function `pass_to_function_with_lvalue_ref_param()`, the pointer is passed to the function `ptr_param_is_lvalue_ref()` as an lvalue reference and is then modified inside that function by member function `reset()`.
- In the function `member_function_modifies_ptr()`, the pointer is modified by member function `reset()`.
- In the function `ptr_is_destination_of_copy_move()`, the pointer is the destination of a copy assignment.

### unique_ptr Used as Function Parameter Type

```
#include <memory>
#include <cassert>
#include <iostream>
#include <string>

static int level = 0;
class A
{
};
void ptr_is_moved_from(
    std::unique_ptr<A> ptr,       // compliant
    const std::unique_ptr<A> ptr1 // non-compliant
)
{

  std::unique_ptr<A> destinationPtr = std::move(ptr);
#if 0
        // compilation error
        destinationPtr = std::move(ptr1); //Cannot be copied
#endif
}

//

void ptr_param_is_lvalue_ref(std::unique_ptr<A> &ptr)
{
  ptr.reset(new A());
}

void ptr_param_is_const_lvalue_ref(const std::unique_ptr<A> &ptr) // compliant
{
  ++level;
  std::cout << std::string(level, '\t') << __FUNCTION__ << "\n";
  std::cout << std::string(level, '\t') << "ptr " << ptr.get() << "\n";
  std::cout << std::string(level, '\t') << __FUNCTION__ << "\n";
  --level;
}

void pass_input_unique_ptr_as_lvalue_reference(
    std::unique_ptr<A> ptr,        // compliant
    const std::unique_ptr<A> ptr1, // non-compliant
    std::unique_ptr<A> ptr2,       // non-compliant
    const std::unique_ptr<A> ptr3  // non-compliant
)
{
  ptr_param_is_lvalue_ref(ptr);

#if 0
        // compilation error
        ptr_param_is_lvalue_ref(ptr1);
#endif

  ptr_param_is_const_lvalue_ref(ptr2); // ptr2 passed as const lvalue reference, lifetime not affected

  ptr_param_is_const_lvalue_ref(ptr3); // ptr3 passed as const lvalue reference, lifetime not affected
}

//

void ptr_modified_by_member_function(
    std::unique_ptr<A> ptr,       // compliant
    const std::unique_ptr<A> ptr1 // non-compliant
)
{
  ptr.release();
#if 0
```

```
        // compilation error
        ptr1.release();
#endif
}

//

void ptr_is_destination_of_copy_move(
    std::unique_ptr<A> ptr,       // compliant
    const std::unique_ptr<A> ptr1 // non-compliant
)
{
  std::unique_ptr<A> gPtr0 = std::make_unique<A>();

  ptr = std::move(gPtr0);
#if 0
        // compilation error
        ptr1 = std::move(gPtr0);

#endif
}
```

In this example, Polyspace reports the use of the unique pointer parameter `const std::unique_ptr<A> ptr1` because the parameter is `const`-qualified and cannot be modified. The lifetime of the parameter is not affected inside these functions and attempts to modify the parameter results in compilation errors.

- Function `ptr_is_moved_from()`.
- Function `pass_input_unique_ptr_as_lvalue_reference()`.
- Function `ptr_modified_by_member_function()`.

Similarly, Polyspace reports the use of the parameters `std::unique_ptr<A> ptr2` and `const std::unique_ptr<A> ptr3` because they are passed to the function `ptr_param_is_const_lvalue_ref()`, which does not affect the lifetime of these parameters.

Polyspace does not report the use of the unique pointer parameter `std::unique_ptr<A> ptr` in these functions because, in each case, the lifetime of the smart pointer is affected:

- In the function `ptr_is_moved_from()`, the unique pointer is moved-from.
- In the function `pass_input_unique_ptr_as_lvalue_reference()`, the pointer is passed to the function `ptr_param_is_lvalue_ref()` as an lvalue reference and is then modified inside that function by member function `reset()`.
- In the function `ptr_modified_by_member_function()`, the pointer is modified by member function `release()`.
- in the function `ptr_is_destination_of_copy_move()`, the pointer is the destination of a move assignment.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2022b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `AUTOSAR C++14 Rule A8-4-12` | `AUTOSAR C++14 Rule A8-4-13`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A8-4-12

A std::unique_ptr shall be passed to a function as: (1) a copy to express the function assumes ownership (2) an lvalue reference to express that the function replaces the managed object.

## Description

### Rule Definition

*A std::unique_ptr shall be passed to a function as: (1) a copy to express the function assumes ownership (2) an lvalue reference to express that the function replaces the managed object.*

### Rationale

You use an `std::unique_ptr` smart pointer to indicate that only one smart pointer owns the managed object. A smart pointer manages the lifetime of a dynamically allocated object and destroys that object when the smart pointer goes out of scope. Pass an `std::unique_ptr` smart pointer to a function in one of these ways:

- If you expect the function to take ownership of the managed object, pass the smart pointer by value . For instance, the `std::unique_ptr` argument of the caller in this code is moved-from and ownership of the `int` object is transferred to an implicitly constructed `std::unique_ptr` smart pointer, which is then passed to the function `func()`.

  ```
  void func(std::unique_ptr<int>) {}
  void otherFunc()
  {
      auto smartPtr = make_unique<int>(1);
      func(std::move(smartPtr)); //caller
  }
  ```
- If you expect the function to replace the object that the smart pointer manages, pass the smart pointer as an lvalue reference. For instance, you might replace the managed object by calling the assignment operator or the `reset()` smart pointer member function. For example, `uniquePtr` is reset inside `func()` and no longer manages the `int` object in this code.

  ```
  void func(std::unique_ptr<int>& smartPtr)
  {
      smartPtr.reset();
  }
  void otherFunc
  {
      auto uniquePtr = make_unique<int>(1);
      func(std::move(uniquePtr); //caller
  }
  ```

  If you do not intend to replace the managed object, pass an lvalue reference to the managed object itself.

You cannot move or reset a `const`-qualified lvalue reference to an `std::unique_ptr` smart pointer. If you do not intend to modify the managed object, pass a `const`-qualified lvalue reference to the managed object instead.

There is no violation of this rule if you pass an rvalue reference to an `std::unique_ptr` smart pointer and the reference is moved into a different `std::unique_ptr` smart pointer inside the function. For instance, `rvalueRefPtr` is passed by rvalue reference to an `std::unique_ptr` smart pointer and then moved into a different smart pointer `std::unique_ptr newPtr`.

```
void func(std::unique_ptr<int>&& rvalueRefPtr)
{
    std::unique_ptr<int> newPtr(std::move(rvalueRefPtr));
}
```

### Polyspace Implementation

Polyspace the use of a parameter passed as an lvalue reference to `std::unique_ptr` if that parameter is not reset.

Polyspace considers the parameter reset in these cases:

- The parameter is used as the destination of a move assignment.
- The parameter is moved-from through a call to `std::move`.
- The parameter is used as the argument of `std::unique_ptr` member functions `reset()`, `swap()`, or `release()`.
- The parameter is used as the argument of any function that takes a non-`const` qualified lvalue reference to an `std::unique_ptr` smart pointer, regardless of the implementation of the callee.

Polyspace also reports the use of `const`-qualified lvalue references to an `std::unique_ptr` smart pointer because you cannot modify the pointer and transfer ownership of the managed object.

Polyspace does not report the use of smart pointers as parameter types in these cases:

- The pointer is used as a parameter of a template function. The parameter type of template functions depends on the instantiation of the function and a fix is not always possible at the template design level. For instance, Polyspace does not report the use of unique pointer `ptr` as parameter in this code.

  ```
  #include <iostream>
  #include <memory>
  #include <cassert>

  template <typename T>
  double XtimesY(T& ptr) // Use of ptr not flagged
  {
    return (ptr->x) * (ptr->y);
    // ...
  }
  struct S
  {
    double x;
    double y;
    S(double x0, double y0) : x{x0}, y{y0} {}
  };
  void func()
  {
    auto a = std::make_unique<S>{100.0, 200.0};
    auto b = new S(100.0, 200.0);
    assert(XtimesY(a) == XtimesY(b));
  }
  ```
- The smart pointer parameter is captured by a lambda function inside the function body. For instance, Polyspace does not report a violation when you use an lvalue reference to unique pointer

`ptr` as a parameter of `func()` because it is captured by a lambda function inside the body of `func()`. Polyspace reports the use of lambda function parameter `up1` because its lifetime is not modified inside the lambda function.

```
#include <memory>

struct S
{
  double x;
  double y;
  S(double x0, double y0) : x{x0}, y{y0} {}
};

void func(std::unique_ptr<S>& ptr)
{ // no defect is detected on 'ptr'
  auto lambdaF =
      [&ptr0 = ptr]            // 'ptr' is captured (by reference)
      (std::unique_ptr<S> up1) // Non-compliant
  {                            // lifetime of 'up1' not affected
    // ...
  };
  auto b = std::make_unique<S>(100.0, 200.0);
  lambdaF(std::move(b));
}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### `std::unique_ptr` Passed to Function

```
#include <memory>
#include <cassert>
#include <iostream>

void func1(std::unique_ptr<int>& uniquePtrParam) // Non-compliant
{
  // uniquePtrParam not reset or modified
}

void func2(const std::unique_ptr<int>& uniquePtrParam) // Non-compliant
{
  // const lvalue reference cannot be modified
}

void func_param_lvalue_ref(std::unique_ptr<int>& uniquePtrParam) // Compliant
{
  std::unique_ptr<int> newPtr = std::make_unique<int>(-8);
  std::swap(newPtr, uniquePtrParam);
}

void func_param_const_lvalue_ref(const std::unique_ptr<int>& uniquePtrParam);

void func3(std::unique_ptr<int>& uniquePtrParam) // Compliant
{
  // uniquePtrParam passed to function with lvalue reference
  // to std::unique_ptr parameter

  func_param_lvalue_ref(uniquePtrParam);
}

void func4(std::unique_ptr<int>& uniquePtrParam) // Non-compliant
{
  // uniquePtrParam passed to function with const lvalue reference
  // to std::unique_ptr parameter. Function cannot modify managed object

  func_param_const_lvalue_ref(uniquePtrParam);
}
```

In this example, Polyspace reports the use of `uniquePtrParam` as a parameter to these functions: v

- `func1()` — The lvalue reference is not modified or reset inside the function.
- `func2()` — The lvalue reference is `const`-qualified and cannot be modified.
- `func4()` — The lvalue reference parameter is passed as an argument to function `func_param_const_lvalue_ref()`, which has a `const`-qualified parameter. This indicates the passed parameter will not be modified inside that function.

If you do not intend to transfer the ownership of the managed object or to reset the `std::unique_ptr` smart pointer, pass an lvalue reference to the managed object itself instead.

Polyspace does not report the use of the `std::unique_ptr` lvalue reference as a parameter of `func_param_lvalue_ref()` because the parameter is used as an argument of `swap()` and the ownership of the managed object is transferred.

Similarly, Polyspace does not report the use of the parameter of `func3()` because it is used as an argument of the function `func_param_lvalue_ref()`. The function `func_param_lvalue_ref()` takes an non-`const` lvalue reference to an `std::unique_ptr` smart pointer as a parameter which indicates it might modify that parameter.

Polyspace does not report the use of the parameter of `func3()` regardless of the implementation of `func_param_lvalue_ref()`.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2022b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `AUTOSAR C++14 Rule A8-4-11`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A8-4-13

A std::shared_ptr shall be passed to a function as: (1) a copy to express the function shares ownership (2) an lvalue reference to express that the function replaces the managed object (3) a `const` lvalue reference to express that the function retains a reference count.

## Description

### Rule Definition

*A std::shared_ptr shall be passed to a function as: (1) a copy to express the function shares ownership (2) an lvalue reference to express that the function replaces the managed object (3) a* `const` *lvalue reference to express that the function retains a reference count.*

### Rationale

You use an `std::shared_ptr` smart pointer to indicate that more than one smart pointer manages and shares ownership of the same object. A smart pointer manages the lifetime of a dynamically allocated object and destroys that object when the smart pointer goes out of scope. Pass an `std::shared_ptr` smart pointer to a function in one of these ways:

- If you expect the function to share ownership of the managed object, pass the smart pointer by value. The `std::shared_ptr` reference count is incremented at the start of the function and decremented at the end of the function. For instance, `smartPtr` is passed to `func()` in this code and that `func()` shares ownership of the managed `int` object until the function ends.

  ```
  void func(std::shared_ptr<int> smartPtr) {
    //Shared ownership of managed object
    //smartPtr until end of func
  }
  ```
- If you expect the function to replace the object that the smart pointer manages on at least one code path, pass the smart pointer by lvalue reference. For instance, you replace the managed object by calling `operator=` or `std::shared_ptr` member function `reset()`.

  For example, the lvalue reference parameter of `func()` is reset inside the function and manages a new `int` object in this code. The function does not share ownership of the original managed object.

  ```
  void func(std::shared_ptr<int>& smartPtr) {
    smartPtr.reset(new int(1));
  }
  ```

  If you do not intend to replace the managed object on at least one code path, pass an lvalue reference to the managed object itself (`void func(int& obj);`).
- If you expect that the function might share ownership of the managed object and copy the parameter to another `std::shared_ptr` smart pointer on at least one execution path, pass the smart pointer by `const` lvalue reference. Passing by `const` lvalue reference avoids unnecessary copies on the other execution paths.

  For instance, the parameter `smartPtr` is not compliant because it is not copied to another `std::shared_ptr` on any execution path in this code. If you do not intend to copy the parameter to another `std::shared_ptr` smart pointer, pass the parameter by `const` lvalue reference to the managed object itself.

```
#include <iostream>
#include <memory>

void func(const std::shared_ptr<int>& smartPtr) { //Non-compliant
  std::cout << *smartPtr << std::endl;
}
```

### Polyspace Implementation

Polyspace reports the use of an `std::shared_ptr` smart pointer parameter passed as an lvalue reference if that parameter is not reset.

Polyspace considers that a parameter is reset in these cases:

- The parameter is used as the destination of a copy or move assignment.
- The parameter is moved-from through a call to `std::move`.
- The parameter is used as an argument to `std::shared_ptr` member functions `reset()` or `swap()`.
- The parameter is used as the argument of any function that takes a non-`const` qualified lvalue reference to an `std::shared_ptr` smart pointer, regardless of the implementation of the callee.

Polyspace also reports the use of an `std::shared_ptr` pointer parameter passed as a `const` lvalue reference if that parameter is not copied to another `std::shared_ptr` pointer along at least one execution path.

The parameter is copied to another `std::shared_ptr` smart pointer if it is used as the argument of one of these functions that take a `const` lvalue reference to `std::shared_ptr` as a parameter:

- `std::static_pointer_cast`
- `std::dynamic_pointer_cast`
- `std::const_pointer_cast`
- `std::reinterpret_pointer_cast`

If you do not intend to copy the parameter to another `shared_ptr` smart pointer, pass a `const` lvalue reference to the managed object instead.

Polyspace does not report the use of smart pointers as parameter types in these cases:

- The pointer is passed as an rvalue reference parameter to a function and the pointer is effectively moved-from inside that function.
- The pointer is used as a parameter of a template function. The parameter type of a template function depends on the instantiation of that function and a fix is not always possible at the template design level. For instance, Polyspace does not report the use of shared pointer `ptr` in this code.

  ```
  #include <iostream>
  #include <memory>
  #include <cassert>

  template <typename T>
  double XtimesY(T& ptr) // Use of ptr not flagged
  {
    return (ptr->x) * (ptr->y);
    // ...
  ```

```
}
struct S
{
  double x;
  double y;
  S(double x0, double y0) : x{x0}, y{y0} {}
};
void func()
{
  auto a = std::make_shared<S>{100.0, 200.0};
  auto b = new S(100.0, 200.0);
  assert(XtimesY(a) == XtimesY(b));
}
```

- The smart pointer parameter is captured by a lambda function inside the function body. For example, Polyspace does not report the use of the lvalue reference to a shared pointer `ptr` as a parameter of `func()` because a lambda function captures the parameter inside the body of `func()`. Polyspace reports the use of the lambda function parameter `up1` because its lifetime is not modified inside the lambda function.

```
#include <memory>

struct S
{
  double x;
  double y;
  S(double x0, double y0) : x{x0}, y{y0} {}
};

void func(std::shared_ptr<S>& ptr)
{ // no defect is detected on 'ptr'
  auto lambdaF =
      [&ptr0 = ptr]            // 'ptr' is captured (by reference)
      (std::shared<S> up1) // Non-compliant
  {                            // lifetime of 'up1' not affected
    // ...
  };
  auto b = std::make_shared<S>(100.0, 200.0);
  lambdaF(std::move(b));
}
```

#### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### `std::shared_ptr` Passed to Function

```
#include <iostream>
#include <memory>

void func1(
    std::shared_ptr<int>& smartPtr,      //Non-compliant
    const std::shared_ptr<int>& constSmartPtr//Non-compliant
)
{
  std::cout << smartPtr;
  auto deleter = std::get_deleter<void (*)(int *)>(constSmartPtr);
}
```

```
void func2(std::shared_ptr<int>& smartPtr){ //Non-compliant
  ++(*smartPtr); // Managed object not replaced
}

struct Base
{
  int a;
  virtual ~Base() {}
};
struct Derived : Base
{
  ~Derived() {}
};
void func3(
    std::shared_ptr<Base>& basePtr,            //Compliant
    const std::shared_ptr<Derived>& derivedPtr //Compliant
)
{
  //derivedPtr is copied to another shared_ptr
  //basePtr is the destination of a copy assigment
  basePtr = std::static_pointer_cast<Base>(derivedPtr);
}
```

In this example, Polyspace reports the use of the `std::shared_ptr` parameters that are passed to these functions:

- `func1()` — The lvalue reference parameter `smartPtr` is not reset and the object managed by `smartPtr` is not replaced. Parameter `constSmartPtr` is noncompliant because the parameter is not copied to another shared pointer. The function `get_deleter` returns the deleter of the smart pointer, but does not create a new shared pointer.
- `func2()` — The `int` object managed by the smart pointer is not replaced.

If you do not intend to modify the managed object of the lvalue reference parameters, pass an lvalue reference to the managed object instead. If you do not intend to copy the `const` lvalue reference parameter to another shared pointer, pass a `const` lvalue reference to the managed object instead.

Polyspace does not report the parameters of `func3()` because:

- The lvalue reference parameter `basePtr` is the destination of a copy assignment.
- The `const` lvalue reference parameter `derivedPtr` is copied to another shared pointer through the call to `std::static_pointer_cast`.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2022b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `AUTOSAR C++14 Rule A8-4-11`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A8-4-14

Interfaces shall be precisely and strongly typed

## Description

### Rule Definition

*Interfaces shall be precisely and strongly typed.*

### Rationale

Interfaces describe the behavior or capabilities of an object. Precisely and strongly typed interfaces specify the purpose and properties of their parameters by using parameters that are custom objects and templates instead of fundamental types. Compare the interfaces in this code snippet:

```
void draw_circle(float R, float x, float y);
void draw_circle(Length R, Position O);
```

Both interfaces represent a function that draws a circle. The first interface uses three floating numbers as input parameters. The second interface uses an object of class `Length` and another object of class `Position` as input parameters.

- The second interface makes it explicit that the first parameter is a length and the second parameter is a position. This interface is easy to understand and use because it highlights the required input parameters and their order for a specific circle. By contrast, you cannot discern the input parameters or their order in the first interface because it is not clear how the three floating numbers relate to the circle.
- The compiler checks the arguments against the input parameter types at compile time. If you put the input parameters of the second interface in the wrong order accidentally, the compiler flags the mismatched parameters at compile time. If all three input parameters of the first interface are floating-point numbers, the compiler cannot check if the input order is correct.
- The definition of the classes `Length` and `Position` can specify the units of these parameters, such as cm or mm. The class definitions can also specify whether these parameters are immutable. When you use fundamental types as input parameters, such specification is difficult.

The first interface is ambiguous because it uses fundamental type input parameters, which can lead to mistakes. Avoid using interfaces that have many fundamental type parameters. Use precisely and strongly typed interfaces instead. Compilers can often optimize such interfaces better than weakly typed interfaces.

When several parameters are related, combine them into a user-defined type. When implementing polymorphic interfaces, use pointers to a common base class instead of pointers to void (`void*`). For generic interfaces, use templates as parameters.

### Polyspace Implementation

In Polyspace, these types are *fundamental types*:

- Integer types, such as `int`, `short`, and `long`
- Floating point types, such as `float` or `double`

- Boolean (bool) types
- Pointers to void (void*)
- Pointers or references to the preceding types
- typedef of the preceding types
- Arrays of the preceding types

In Polyspace, enumerations or enums are not fundamental types. Polyspace flags an interface if its input parameters include any of the following:

- One or more void related types
- Two or more bool related types
- Three or more identical fundamental types

You can use alternatives such as comments or parameter names to clarify an interface definition. In such cases, you can justify the Polyspace result by using comments in your result or code. See "Address Results in Polyspace User Interface Through Bug Fixes or Justifications"

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Using Interfaces with Many Fundamental Type Parameters

Polyspace flags interfaces when their input parameters include:

- Two or more bool related types
- Three or more identical fundamental types

Consider the interfaces in this code:

```
#include <cstdint>
#include <chrono>

void Sleep(std::uint32_t duration);//Compliant

void SetProperty(bool Status);//Compliant

void SetAlarm(std::uint32_t year, std::uint32_t month, //Noncompliant
              std::uint32_t day, std::uint32_t hour,
              std::uint32_t minute, std::uint32_t second);

//Compliant
void StartClock(std::chrono::system_clock::time_point const& when);
typedef struct {
    int a, b, c, d;
} Point;
void Triangle(float a, float b, float c);//Noncompliant
void Rectangle(Point a, Point b, Point c, Point d); //Compliant

main()
```

```
{
    //...
}
```

- Polyspace flags the interfaces that use three or more fundamental type variables as input parameters, such as `SetAlarm()` and `Triangle()`.
- The interfaces `StartClock()` and `Rectangle()` use precise and strongly typed input parameters. Polyspace does not flag these interfaces.
- Polyspace does not flag an interface that has less than three fundamental type input parameters or less than two `bool` type input parameters.

### Avoid Using Pointer to Void (void*)

Polyspace flags interfaces when their input parameters include one or more pointers to void (`void*`). Consider the interfaces in this code:

```
#include <cstdint>

class A{
    //...
};
class B:public A{
    //...
};
class C:public A{
    //...
};

void polymorphic_function(void*);//Noncompliant
void polymorphic_function(A*);//Compliant

void printArray(void* Array);//Noncompliant
template <typename T>
void printArray (T* Array);//Compliant

main(){
    //...
}
```

All pointer types implicitly convert to `void*`, which is a weak and under-qualified type. Avoid using `void*` pointers.

- To implement polymorphic interfaces, use pointers to base classes such as `A*` instead of `void*`.
- To implement generic interfaces, use templates such as `T*` instead of `void*`.

## Check Information

**Group:** Declarators
**Category:** Required, Non-automated

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A8-5-0

All memory shall be initialized before it is read

## Description

### Rule Definition

*All memory shall be initialized before it is read.*

### Rationale

The C++ standard does not specify what the value of an uninitialized memory can be. This value is unpredictable and can be different every time the program runs. Reading and using the value of an uninitialized memory results in unexpected behavior.

### Polyspace Implementation

Polyspace reports a violation of this rule if your code contains these issues:

- `Non-initialized variable`
- `Member not initialized in constructor`
- `Non-initialized pointer`

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Variable Read Before Initialization

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val; //Noncompliant

}
```

### Member Not Initialized in Constructor

In this example, if flag is not 0, the member _c is not initialized.

The defect appears on the closing brace of the constructor. Following are some tips for navigating in the source code:

- On the **Result Details** pane, see which members are not initialized.
- To navigate to the class definition, right-click a member that is initialized in the constructor. Select **Go To Definition**. In the class definition, you can see all the members, including those members that are not initialized in the constructor.

```
class MyClass {
public:
    explicit MyClass(int);
private:
    int _i;
    char _c;
};

MyClass::MyClass(int flag) {
    if(flag == 0) {
        _i = 0;
        _c = 'a';
    }
    else {
        _i = 1;
    }
}//Noncompliant
```

### Pointer Dereferenced Before Initialization

If prev is not NULL, the pointer pi is not assigned an address. However, pi is dereferenced on every execution paths, irrespective of whether prev is NULL or not.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j; //Noncompliant

    return pi;
}
```

## Check Information

**Group:** Declarators
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A8-5-1

In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition

## Description

### Rule Definition

*In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A8-5-2

Braced-initialization {}, without equals sign, shall be used for variable initialization

## Description

### Rule Definition

*Braced-initialization {}, without equals sign, shall be used for variable initialization.*

### Rationale

Braced initialization:

```
classType Object{arg1, arg2, ...};
```

is less ambiguous than other forms of initialization. Braced initialization has the following advantages:

- Prevents implicit narrowing conversions such as from `double` to `float`.
- Avoids the ambiguous syntax that leads to the problem of most vexing parse.

  For instance, from the declaration:

  ```
  ResourceType aResource();
  ```

  It is not immediately clear if `aResource` is a function returning a variable of type `ResourceType` or an object of type `ResourceType`.

  For more information, see `Ambiguous declaration syntax`.

The rule also forbids the use of = sign for initialization because the = sign can give the impression that an assignment or copy constructor is invoked even in situations when it is not.

### Polyspace Implementation

In general, the checker flags initializations of an object `obj1` of data type `Type` using these formats:

- `*Type* obj1 = obj2;`
- `*Type* obj1(obj2);`

The checker allows an exception for these cases:

- Initialization of variables with type `auto` using a simple assignment to a constant, a variable, a lambda expression, a standard initializer list or a function call.
- Initialization of reference types using a simple assignment
- Declarations with global scope using the format *`Type`* `a()` where *`Type`* is a class type with default constructor. The analysis interprets `a` as a function returning the type *`Type`*.
- Loop variable initialization in OpenMP parallel `for` loops, that is, in `for` loop statements that immediately follow `#pragma omp parallel for`

The checker is enabled only if you specify a C++ version of C++11 or later. See `C++ standard version (-cpp-version)`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Braced and Nonbraced Initialization

```
class ResourceType {
      int memberOne;
      int memberTwo;
    public:
      ResourceType() {memberOne = 0; memberTwo = 0;}
      ResourceType(int m, int n) {memberOne = m; memberTwo = n;}
      ResourceType(ResourceType &anotherResource) {
          memberOne = anotherResource.memberTwo;
          memberTwo = anotherResource.memberOne;
      }
};

void func() {
    ResourceType aResourceOne(); //Noncompliant
    ResourceType aResourceTwo(1, 2); //Noncompliant
    ResourceType aResourceThree = {1,2};   //Noncompliant

    ResourceType aResourceFour{1,2}; //Compliant

}
```

In this example, the function `func` declares four objects of type `ResourceType`. Only the declaration of `aResourceFour` does not violate this rule.

The declarations of `aResourceOne`, `aResourceTwo` and `aResourceThree` violate the rule. In particular:

- The declaration of `aResourceOne` suffers from the problem of most vexing parse. It is not clear whether `aResourceOne` is an object of type `ResourceType` or a function returning an object of type `ResourceType`.
- The declaration of `aResourceThree` seems to suggest that the copy constructor `ResourceType(ResourceType &)` is invoked for initialization. The copy constructor initializes the data member `memberOne` to 2 and `memberTwo` to 1. However, the constructor `ResourceType(int, int)` is invoked. This constructor initializes the data member `memberOne` to 1 and `memberTwo` to 2.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Ambiguous declaration syntax` | `Variable shadowing` | `Non-initialized variable` | `Write without a further read` | `Improper array initialization` | `Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A8-5-3

A variable of type auto shall not be initialized using {} or ={} braced-initialization

## Description

### Rule Definition

*A variable of type auto shall not be initialized using {} or ={} braced-initialization.*

### Rationale

Type deduction for `auto` has a counter-intuitive result when the initialization uses braces. The deduced type is `std::initializer_list<>` instead of the type that you might guess from the initializer.

For instance, the definition:

```
auto x{1};
```

results in the type of `x` being `std::initializer_list<int>` instead of `int`. Some compilers deduce an `int` type from this definition, but the behavior is not uniform across compilers.

### Polyspace Implementation

The checker flags variable definitions that use the type `auto` if the variable is initialized using the `{}` or `={}` braced initialization.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `auto` in Braced Initialization

```
#include<initializer_list>

void func() {
    auto aVar{1}; //Noncompliant
    auto anotherVar(1); //Compliant
    int aThirdVar{1}; //Compliant

    auto aVarList{1,2,3}; //Noncompliant
    std::initializer_list<int> anotherVarList{1,2,3}; //Compliant
}
```

In this example, the rule is violated when the `auto` type is used with braced initialization. Instead of `auto`, an explicit type specification is preferred. Alternatively, the initialization can use parenthesis `()`, which ensures the expected type deduction.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A8-5-4

If a class has a user-declared constructor that takes a parameter of type std::initializer_list, then it shall be the only constructor apart from special member function constructors

## Description

### Rule Definition

*If a class has a user-declared constructor that takes a parameter of type std::initializer_list, then it shall be the only constructor apart from special member function constructors.*

### Rationale

If a class contains a constructor that takes a parameter of type `std::initializer_list` and another constructor with parameters, braced initializations such as:

```
classType obj {0,1}
```

Can lead to confusion about which of the two constructors is invoked. Compilers prefer the constructor with the `std::initializer_list` parameter, but developers might expect otherwise.

### Polyspace Implementation

The checker flags class definitions that contain a constructor whose first parameter is of type `std::initializer_list` and also contains another constructor (excluding the special member function constructors). The rule violation is followed by events that point to the location of the other constructors that might lead to confusion with the `std::initializer_list` constructor.

A class definition with an `std::initializer_list`-parameter constructor that does not violate this rule has only the default, copy and move constructors (and copy and move assignment operators). If you cannot avoid a second constructor with parameters, you can justify this rule violation. In that case, for initializing with a list, use a syntax such as:

```
classType obj ({0,1})
```

so that it is clear that the `std::initializer_list`-parameter constructor is invoked. For initializing with the other constructors, use a syntax such as:

```
classType obj (0,1)
```

Both invocations are exceptions to Rule A8-5-2, which generally flags initializations with `()`, but allows such initializations for classes with a mix of `std::initializer_list`-parameter constructor and other constructors.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant and Compliant Definitions of Classes with `std::initializer_list`-Parameter Constructors

```
#include <cstdint>
#include <initializer_list>
#include <vector>

class entrySizes {                  //Noncompliant
    public:
       entrySizes()=default;
       entrySizes(std::size_t aCurrentSize, std::size_t aLastSize):
              currentSize{aCurrentSize}, lastSize{aLastSize} {}
       entrySizes(std::initializer_list<std::size_t> sampleEntry):
              currentSize{sampleEntry.size()}, lastSize{sampleEntry.size()} {}
    private:
       std::size_t currentSize;
       std::size_t lastSize;
};

class recordSizes {                 //Compliant
    public:
       recordSizes()=default;
       recordSizes(std::initializer_list<std::size_t> sampleRecord):
              currentSize{sampleRecord.size()}, lastSize{sampleRecord.size()} {}
    private:
       std::size_t currentSize;
       std::size_t lastSize;
};


void createEntry() {
    entrySizes defaultEntrySize{};
    entrySizes stdEntrySize{0,1};
       //Calls entrySizes(std::initializer_list<std::size_t> ),
       //but developer might expect otherwise
    entrySizes expectedEntrySize({0,1});
       //Calls entrySizes(std::initializer_list<std::size_t> ),
       //but developer might expect otherwise
    entrySizes typicalEntrySize(1,1);
        //Calls entrySizes(std::size_t, std::size_t)
}

void createRecord() {
    recordSizes defaultRecordSize{};
    recordSizes stdRecordSize{0,1};
       //Calls recordSizes(std::initializer_list<std::size_t> )
}
```

In this example, the class `entrySizes` contains two user-defined constructors, one with an `std::initialize_list` parameter and a second one with two `size_t` parameters. The presence of two constructors can lead to developer confusion as shown in the `createEntry` function. In case you want to retain the current class definition and justify the rule violation, the `createEntry` function also shows a cleaner way to invoke the `std::initialize_list`-parameter constructor.

The class `recordSizes` does not violate the rule since it does not contain another constructor other than the default constructor and the constructor with the `std::initialize_list` parameter.

## Check Information

**Group:** Declarators
**Category:** Advisory, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A9-3-1

Member functions shall not return non-constant "raw" pointers or references to private or protected data owned by the class

## Description

### Rule Definition

*Member functions shall not return non-constant "raw" pointers or references to private or protected data owned by the class.*

### Rationale

Returning a nonconstant pointer or reference to private or protected class-owned data enables clients to externally access and modify the state of the object without an interface. Such access without an explicit interface might bypass the private/protected data access hierarchy of the class, which might result in unexpected behavior and lead to bugs.

This rule applies to data that is owned by the class. Nonconstant handles to objects that are shared between different classes might be returned. Classes that mimic smart pointers and containers do not violate this rule.

### Polyspace Implementation

The checker flags a rule violation only if a member function returns a non-`const` pointer or reference to a nonstatic data member. The rule does not apply to static data members.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Returning Non-Constant Raw Pointers to Private Data

```
#include <cstdint>
#include <memory>
#include <utility>

class A
{
  public:
    explicit A(std::int32_t number) : x(number) {}
    std::int32_t&
    GetX() noexcept // Noncompliant
    {
      return x;
    }

  private:
    std::int32_t x;
};

void Fn1() noexcept
{
  A a{10};
  std::int32_t& number = a.GetX();
  number = 15; // External modification of private class data
}
```

In this example, the `class A` member function `GetX()` returns a non-constant raw pointer to `x`, which is private data owned by `class A`. Polyspace flags this implementation as noncompliant. `Fn1()` demonstrates the issues of `a.GetX()` returning a non-constant raw pointer to private class data, which is then stored and modified by number. The class has no control over changes to its own private data member, which might lead to unexpected behavior.

#### Compliant: Return Shared Smart Pointer Variables

```
#include <cstdint>
#include <memory>
#include <utility>
class B
{
  public:
    explicit B(std::shared_ptr<std::int32_t> ptr) : sharedptr(std::move(ptr)) {}
    std::shared_ptr<std::int32_t> GetSharedPtr() const noexcept // Compliant
    {
      return sharedptr;
    }

  private:
    std::shared_ptr<std::int32_t> sharedptr;
};

void Fn2() noexcept
{
  std::shared_ptr<std::int32_t> ptr = std::make_shared<std::int32_t>(10);
  B b1{ptr};
  B b2{ptr};
  *ptr = 50; // External modification of ptr which shared between b1 and b2
  // instances
  auto shared = b1.GetSharedPtr();

  *shared = 100;   // External modification of ptr which shared between b1 and
  // b2 instances
}
```

In this example, the `class B` function `GetSharedPtr()` returns a smart pointer variable that is shared between the instances `b1` and `b2`. Polyspace does not flag this implementation as noncompliant.

#### Compliant: Return Constant References

```
#include <cstdint>
#include <memory>
#include <utility>
class C
{
  public:
    explicit C(std::int32_t number)
      : ownedptr{std::make_unique<std::int32_t>(number)}
    {
    }
    const std::int32_t& GetData() const noexcept // Compliant
    {
      return *ownedptr;
    }

  private:
    std::unique_ptr<std::int32_t> ownedptr;
};
void Fn3() noexcept
{
  C c{10};
  const std::int32_t& data = c.GetData();
  // data = 20; // Cannot modify data, it is a const reference
}
```

In this example, `GetData()` returns a constant reference. You cannot modify private class-data by using this member function. Polyspace does not flag this implementation as noncompliant.

## Check Information

**Group:** Classes
**Category:** Required, Partially automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A9-5-1

Unions shall not be used

## Description

### Rule Definition

*Unions shall not be used.*

### Rationale

Using unions to store a value might result in misinterpretation of the value and lead to undefined behavior. For instance:

```
union Data{
    int i;
    double d;
};
void bar_int(int);
void bar_double(double);
void foo(void){
    Data var;
    var.d = 3.1416;
    bar_int(var.d);//Undefined Behavior
}
```

In the call to `bar_int`, the `double` data in the union is misinterpreted as an `int`, which is undefined behavior. Compilers might react to this misinterpretation differently depending on their implementation. To avoid undefined behaviors, do not use a `union`.

In some cases, use of unions might be necessary to increase efficiency. In such cases, use unions after documenting the relevant implementation-defined compiler behaviors. In the preceding case, before using a `union`, consult the manual of the compiler that you use and document how the compiler reacts to interpreting a `double` as an `int`.

As an exception, use of tagged union is allowed until `std::variant` becomes available in the C++ standard library (C++17).

### Polyspace Implementation

Polyspace flags the declaration of a `union`. You might consider the use of `union` necessary or acceptable in your code. In such cases, justify the violation by annotating the result or by using code comments. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using union

```
#include <iostream>

union Pi{ //Noncompliant
    int i;
    double d;
};

void foo(void){

    std::cout << std::endl;

    Pi pi;
    pi.d = 3.1416;// pi holds a double
    std::cout << "pi.d: " << pi.d << std::endl;
    std::cout << "pi.i: " << pi.i << std::endl;      // Undefined Behavior

    std::cout << std::endl;

    pi.i = 4;     // pi holds an int
    std::cout << "pi.i: " << pi.i << std::endl;
    std::cout << "pi.d: " << pi.d << std::endl;      // Undefined Behavior

    std::cout << std::endl;

}
```

In this example, the union Pi contains a double and an int. In the code, a double is misinterpreted as an int and vice versa by using the union. These misinterpretations are undefined behaviors and might lead to bugs and implementation dependent code behavior. Polyspace flags the union declaration.

## Check Information

**Group:** Classes
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

Check AUTOSAR C++ 14 (-autosar-cpp14)

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A9-6-1

Data types used for interfacing with hardware or conforming to communication protocols shall be trivial, standard-layout and only contain members of types with defined sizes

## Description

### Rule Definition

*Data types used for interfacing with hardware or conforming to communication protocols shall be trivial, standard-layout and only contain members of types with defined sizes.*

### Rationale

When interfacing with hardware or when conforming to communication protocols, the layout of the data type you use must be consistent and standardized. For these tasks, use data types with layout and size that are not dependent on your hardware or software.

Types with well defined sizes are listed in the header `<cstdint>`. Examples of such data type includes types such as `int16_8` and `int32_t`. This list excludes types such as `bool`, `wchar_t`, and pointers. Avoid using excluded types when interfacing with hardware or when conforming to communication protocols.

Using enumerations are allowed only when the underlying data types of the `enum` has a well defined size. Using bitfields for these tasks are allowed only when the underlying type has a well defined size,

### Polyspace Implementation

Polyspace raises a violation of this rule if you use a data type other then these two in a bit field:

- Enumeration
- Unsigned integral type

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use `enum` and Unsigned Integral Types in Bitfields

```
#include <cstdint>
enum class E1: std::uint8_t {
    E11,
    E12,
    E13
};
enum class E2{
    E21,
    E22,
    E23
};
class hwInterface{
public:
    std::int32_t A:2; //Noncompliant- signed integer
    std::uint8_t B:2U; //Compliant
    E1 field1:2; //Compliant

    char D:2;//Noncompliant
    E2 field2:2; //Compliant
};
```

In this example, Polyspace flags the use of any types other than `enums` and unsigned integers in bitfields. For instance, the use of `std::int32_t` and `char` is flagged. Polyspace does not flag the use of `std::uint8_t` or enums such as `E1` or `E2`.

## Check Information

**Group:** Classes
**Category:** Required, Partially automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A10-1-1

Class shall not be derived from more than one base class which is not an interface class

## Description

### Rule Definition

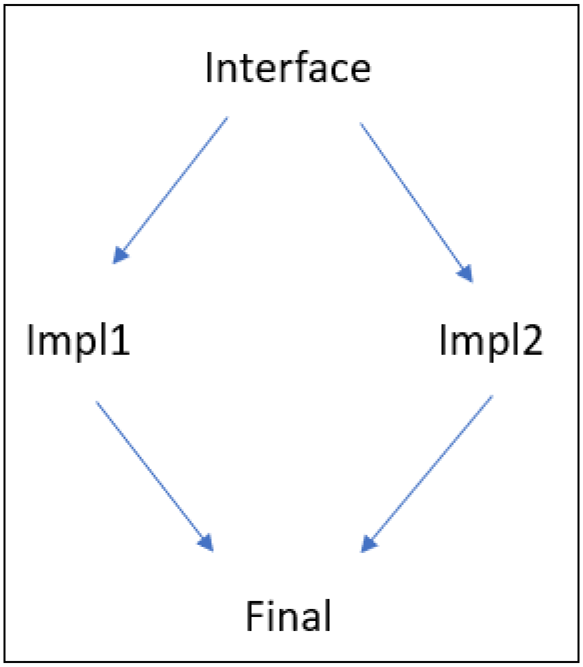
*Class shall not be derived from more than one base class which is not an interface class.*

### Rationale

If a class inherits from multiple non-interface classes, the class essentially has access to multiple implementations. Maintaining the code can be difficult.

When a class inherits from multiple non-interface classes, there is a likelihood that the same member function exists in those base classes and must be overridden in the derived class. The likelihood increases when those base classes themselves inherit from a common base class (diamond structure).

Suppose, an interface class `Interface` has two concrete implementations, `Impl1` and `Impl2`, and a class `Final` derives from both implementations. The class hierarchy has this diamond structure.



The following issues can occur:

- *Overrides required in final derived class for disambiguation*:

  Both implementations `Impl1` and `Impl2` have a copy of all methods of the class `Interface`. To disambiguate which copy can be called through a `Final` object, you typically create yet another override of all methods in the `Final` class where you call both copies explicitly using the scope resolution operator `::` (or one copy, if you choose). See example below.

Each time you add a new pure virtual function to the class `Interface`, you have to not only create implementations in the immediate derived classes but also keep track of the entire class hierarchy and create overrides of those implementations in the class `Final`.

If the original class `Interface` is not an interface class, the problem is even more acute. Unless the inheritances are virtual, two copies of the methods of `Interface` are *implicitly* made in `Impl1` and `Impl2` (the diamond problem).

- *Final derived class responsible for initializing all classes in hierarchy*:

  To avoid double initializations in multiple inheritance, the C++ standard requires that you call the constructors of all previous classes in the most derived class.

  In the preceding example, the `Final` class constructor not only has to call the constructors of `Impl1` and `Impl2` but also the constructor of their parent class `Interface`. You have to trace beyond the immediate parents to determine which constructors to call in the final derived class.

These problems disappear if multiple inheritances are restricted to situations where a class can derive from multiple classes but only one of them can be a non-interface class. An interface class is a class that has only pure virtual functions and data members that are compile-time constants (static, `contexpr`-s). The class has no state and its sole purpose is to be implemented by derived classes.

Multiple inheritance was designed for situations where a class extends one concrete implementation but also implements other ideas represented by interface classes. Other uses of multiple inheritance can lead to maintenance hazards.

### Polyspace Implementation

The checker flags multiple inheritances where more than one base class is a non-interface class.

An interface class is one that has only pure virtual functions and data members that are compile-time constants (static, `contexpr`-s). Any constructor or destructor is set to `=default` or `=delete`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Multiple Inheritance from Non-interface Classes

```
class Interface {
    public:
    virtual void setVal()=0;
};

class Impl1: public Interface{
    int val1;
public:
    void setVal() {
        val1 = 0;
    }
};

class Impl2: public Interface{
    int val2;
public:
    void setVal() {
        val2 = 0;
    }
};

class Final: public Impl1, public Impl2 { //Noncompliant
public:
    void setVal() {
        Impl1::setVal();
        Impl2::setVal();
    }

};

void main() {
    Final finalObj;
    finalObj.setVal();
}
```

In this example, the class final derives from classes `Impl1` and `Impl2`. Both classes `Impl1` and `Impl2` have data members that are not compile-time constants and member functions that are not pure virtual functions. Therefore, the classes are non-interface classes. Inheriting from two non-interface classes causes a coding rule violation.

## Check Information

**Group:** Derived classes
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A10-2-1

Non-virtual public or protected member functions shall not be redefined in derived classes

## Description

### Rule Definition

*Non-virtual public or protected member functions shall not be redefined in derived classes.*

### Rationale

When a nonvirtual public or protected member function is redefined in a derived class, the new definition in the derived class hides the definition in the base class instead of overriding it. When functions are hidden in the derived class, you cannot implement a common interface to handle different classes of the same hierarchy, resulting in unnecessary complexity and error. Such behavior might be unexpected and lead to bugs that are difficult to resolve.

Redefinitions of functions from private inheritance or functions that are private in the base class are not affected by this rule.

### Polyspace Implementation

Polyspace flags redefinitions of non-virtual member functions in a derived class. Polyspace does not raise this defect on destructors.

To justify a redefinition that you deem as acceptable, use annotations. See “Annotate Code and Hide Known or Acceptable Results”

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Redefining Public or Protected Member Functions in Derived Classes

```
class A
{
  public:
    virtual ~A() = default;
    void F() noexcept {}
    virtual void G() noexcept {}
  private:
    void H() noexcept;
};

class B : public A
{
  public:
    void F() noexcept {} //Noncompliant
    void G() noexcept override {} //Compliant
};
```

In this example, the `A::F()` function is a non-virtual public member function that is hidden by the B class. Hiding `A::F()` prevents the use of polymorphic interfaces, so Polyspace flags the redefinition in `B::F()` as noncompliant. The `A::G()` function is virtual and is overridden (rather than hidden) in `B::G()`, so Polyspace does not flag this implementation as noncompliant.

## Check Information

**Group:** Derived Classes
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A10-3-1

Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final

## Description

### Rule Definition

*Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final.*

### Rationale

Virtual functions implement polymorphic behavior in a class hierarchy. Once you declare a function as `virtual` in a base class, all instances of the function with an identical parameter list in the derived classes override the base function implicitly. If you rely on this implicit action by the compiler for implementing polymorphic functions, it can lead to errors. For instance:

- A function can become inadvertently `virtual` because its signature matches a virtual function in the base class.
- A function can become inadvertently non-virtual because there are differences in the parameter list.

Implicitly declaring virtual functions can also make the code hard to read.

To avoid inadvertent errors and to enhance readability, use the specifiers `virtual`, `override`, or `final` to explicitly define virtual or overriding functions. Because using more than one of these specifiers in a declaration is either redundant or a source of error, use exactly one of these specifiers:

- Only `virtual` to declare a new virtual function.
- Only `override` to declare a non-final overriding function of a virtual function.
- Only `final` to declare a final overriding function of a virtual function.

### Polyspace Implementation

Polyspace flags declaration of virtual functions if:

- The declaration uses none of the specifiers.
- The declaration uses more than one of the specifiers.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use Exactly One Specifier to Declare Virtual Functions

```
#include<cstdint>
class Base
```

```
{
public:
    virtual void F() noexcept = 0;                  // Compliant
    virtual void G() noexcept final = 0;            // Noncompliant
    virtual void H() noexcept final {}              // Noncompliant
    virtual void J() noexcept {}                    // Compliant
    virtual void K() noexcept {}                    // Compliant
    virtual ~Base() {}                              // Compliant
    virtual void M() noexcept {}                    // Compliant
    virtual void Z() noexcept {}                    // Compliant
    virtual void X() throw() {}                     // Compliant
    virtual void Y() noexcept {}                    // Compliant
};

class Derived : public Base
{
public:
    ~Derived() {}                                   // Noncompliant
    virtual void F() noexcept override {}           // Noncompliant
    void K() noexcept override final {}             // Noncompliant
    virtual void M() noexcept {}                    // Compliant
    void Z() noexcept override {}                   // Compliant
    void J() noexcept {}                            // Noncompliant
    void J(int) noexcept {}                         // Compliant
    virtual void X() throw() final {}               // Noncompliant
    virtual void Y() noexcept override final {}     // Noncompliant
};
class DD: public Derived{
//    void J(int) noexcept override{}       //Compilation error
};
main(){
    //...
}
```

- The destructor of the derived class `~Derived()` is a virtual function. Its declaration violates this rule because the declaration contains none of the three specifiers for virtual functions.
- The declaration of the pure virtual function `Base::G()` also violates this rule because the declaration contains both `virtual` and `final` as specifiers. A pure virtual function that is also specified as `final` is redundant.
- The declaration of the virtual function `Derived::J()` violates this rule because `Derived::J()` implicitly overrides the virtual function `Base::J()` without using the specifier `override`.
- The declarations of the virtual functions `Derived::X()` and `Derived::Y()` violate this rule because their declarations use more than one specifier.

The declaration of the function `DD::J(int)` produces a compilation error because `DD::J(int)` is trying to override `Derived::J(int)`. Because `Derived::J(int)` has a different signature than `Base::J()`, perhaps by error, `Derived::J(int)` is no longer a virtual function. Attempting to override `Derived::J(int)` by `DD::J(int)` results in a compilation error. Using exactly one specifier in the declaration of virtual functions can help detect errors.

## Check Information

**Group:** Derived classes
**Category:** Required, Automated

## Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A10-3-2

Each overriding virtual function shall be declared with the override or final specifier

## Description

### Rule Definition

*Each overriding virtual function shall be declared with the override or final specifier.*

### Rationale

Virtual functions implement polymorphic behavior in a class hierarchy. Once you declare a function as `virtual` in a base class, all instances of the function with an identical parameter list in the derived classes override the base function implicitly. If you rely on this implicit action by the compiler for implementing polymorphic functions, it can lead to errors. For instance:

- A function can become inadvertently `virtual` because its signature matches a virtual function in the base class.
- A function can become inadvertently non-virtual because there are differences in the parameter list.

Implicitly declaring overriding virtual functions can also make the code hard to read.

To avoid inadvertent errors and to enhance readability, use the specifiers `override` and `final` explicitly in every declaration of overriding functions.

### Polyspace Implementation

Polyspace flags the declarations of `virtual` functions if all of these statements are true:

- The function is in a derived class.
- The signature of the function matches the signature of a virtual function in the base class.
- The declaration of the function lacks the specifier `override` or `final`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Declare Overriding Virtual Functions by Using `override` or `final` Specifier

```
#include <cstdint>
class Base
{
public:
    virtual ~Base() {}
    virtual void F() noexcept = 0;
    virtual void G() noexcept {}
    virtual void Z() noexcept {}
    virtual Base& operator+=(Base const& oth) = 0;
};
class Derived1 : public Base
{
public:
    ~Derived1() override {}                           //Compliant
    void F() noexcept{}                               //Noncompliant
    virtual void G() noexcept {}                      //Noncompliant
    void Z() noexcept override {}                     // Compliant
    Derived1& operator+=(Base const& oth) override  // Compliant
    {
        return *this;
    }
};
class Derived2 : public Base
{
public:
    ~Derived2() {}                                    //  Noncompliant
    void F() noexcept override {}                     //  Compliant
    void G() noexcept override {}                     //  Compliant
    void Z() noexcept override {}                     //  Compliant
    Derived2& operator+=(Base const& oth)             //  Noncompliant
    {
        return *this;
    }
};
class Derived3 : public Base
{
    void F() noexcept override;                       // Compliant
};

main(){

}
```

- The declaration of the function `Derived::F()` is flagged because its signature matches the signature of `Base::F()` and its declaration does not contain `override` or `final`.
- The declaration of the function `Derived::G()` is flagged because its signature matches the signature of `Base::G()` and its declaration does not contain `override` or `final`, even though the declaration uses the specifier `virtual`.
- The declaration of the function `Derived3::F()` in class `Derived3` is not flagged because the declaration uses the specifier `override`.

## Check Information

**Group:** Derived classes
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A10-3-3

Virtual functions shall not be introduced in a final class

## Description

### Rule Definition

*Virtual functions shall not be introduced in a final class.*

### Rationale

Declaring a function as `virtual` indicates that you intend to override the function in a derived class with a different implementation. The same function can then interact differently with different classes of a hierarchy. When you explicitly specify a class as `final`, you cannot derive a class from it. Because you cannot derive classes from a `final` class, do not introduce virtual functions in a `final` class. Specify all virtual functions in a `final` class by using the specifier `final`.

### Polyspace Implementation

Polyspace flags the declaration of virtual functions in a `final` class. Polyspace does not flag virtual functions in a final class that uses the specifiers `override final` or `virtual final`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Specify Virtual Function in `final` Classes by Using `final`

```
#include <cstdint>
class Base
{
public:
    virtual ~Base() = default;
    virtual void F() noexcept = 0;
    virtual void G() noexcept {/*...*/}
    virtual void Y() noexcept {/*...*/}
};
class Derived final : public Base
{
public:

    void G() noexcept override{/*...*/} //Noncompliant
    virtual void Z() noexcept{/*...*/}  //Noncompliant
    virtual void H() noexcept = 0;    //Noncompliant
    void F() noexcept final{/*...*/}    //Compliant
    void Y() noexcept override final{/*...*/}    //Compliant
};
```

The functions `Derived::G()`, `Derived::Z()`, and `Derived::H()` are `virtual` functions that are not specified as `final`. Their declarations indicate that some functions in a derived class might override these functions. The class `Derived` is specified as `final`. That is, there are no derived classes from this class. The declarations of `Derived::G()`, `Derived::Z()`, and `Derived::H()` are inconsistent with the declaration of their class `Derived`. Polyspace flags the declarations of the functions. The functions `Derived::F()` and `Derived::Y()` are declared as `final`. These declaration comply with this rule.

### Check Information

**Group:** Derived classes
**Category:** Required, Automated

## Version History

**Introduced in R2020a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A10-3-5

A user-defined assignment operator shall not be virtual

## Description

### Rule Definition

*A user-defined assignment operator shall not be virtual.*

### Rationale

Defining an assignment operator as `virtual` in a base class indicates that you want to override it in the derived classes. Overriding the assignment operator in derived classes can lead to undefined behavior and run-time errors. Consider this code snippet where a virtual assignment operator is overridden in two derived classes.

```
class Base {public:
    virtual Base& operator=(Base const& oth) = 0;
    //...
};
class Derived public: Base{ public:
    Derived& operator=(Base const& oth) override{/*...*/}
    //...
};
class Derived2 public: Base{public:
    Derived2& operator=(Base const& oth) override{/*...*/}
    //...
};
main(){
    Derived d1;
       Derived2 d2;
    d1 = d2;
}
```

Because `Derived::operator=` and `Derived2::operator=` overrides `Base::operator=`, their parameter lists must be identical.

- `Derived::operator=` takes reference to a `Base` object as input and returns a reference to `Derived`.
- `Derived2::operator=` takes reference to a `Base` object as input and returns a reference to `Derived2`.

The `Derived::operator=` accepts references to both `Base` and `Derived` class objects because references to derived classes are type-compatible with their base classes. Similarly, the `Derived2::operator=` also accepts references to both `Base` and `Derived2` class objects. Assigning a `Derived` object to a `Derived2` object in `d1=d2` produces no compilation error. The objects `d1` and `d2` are unrelated. Assigning, copying, or moving operations between such unrelated objects are undefined and can lead to run-time errors.

To avoid undefined behavior and run-time errors, keep user-defined assignment operators as non-virtual. This rule applies to these operators:

- Assignment
- Copy and move assignment
- All compound assignment

**Polyspace Implementation**

Polyspace flags the declaration of any virtual assignment operators in a base class.

**Troubleshooting**

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Defining Assignment Operators as `virtual`

This example shows how Polyspace flags `virtual` assignment operators.

```
#include <cstdint>
class Base
{
  public:
    virtual Base& operator=(Base const& oth) = 0;   // Noncompliant
    virtual Base& operator+=(Base const& rhs) = 0; // Noncompliant
};
class Derived : public Base
{
  public:
    Derived& operator=(Base const& oth) override
    {
      return *this;
    }
    Derived& operator+=(Base const& oth) override
    {
      return *this;
    }
    Derived& operator-=(Derived const& oth) // Compliant
    {
      return *this;
    }
};
class Derived2 : public Base
{
  public:

    Derived2& operator=(Base const& oth) override
    {
      return *this;
    }
    Derived2& operator+=(Base const& oth) override
    {
      return *this;
    }
    Derived2& operator-=(Derived2 const& oth)   // Compliant
    {
      return *this;
    }
};
/*
*/
void Fn() noexcept
{
  Derived b;
  Derived2 c;
  b = c;
  b += c;
  c = b;
  c += b;
  // b -= c; // Compilation error
  // c -= b; // Compilation error

}
```

The classes `Derived` and `Derived2` are derived from `Base`. In the `Base` class, the assignment operators `Base::operator=` and `Base::operator+=` are declared as virtual. None of the following cause compilation errors:

- You can assign the `Derived` object `b` to `Derived2` object `c` and vice versa.
- You can add the `Derived` object `b` to `Derived2` object `c`. You can assign the result to either `b` or `c`.

Because `b` and `c` are unrelated objects, all of the preceding behaviors are undefined and can cause run-time errors. Declaring the `Base::operator=` and `Base::operator+=` as `virtual` eventually lead to the undefined behaviors. Polyspace flags these virtual assignment operators.

The declaration of `Base::operator-=` is non-virtual. Operations such as `b-=c` and `c-=b` cause compilation errors.

## Check Information

**Group:** Derived classes
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A10-4-1

Hierarchies should be based on interface classes

## Description

### Rule Definition

*Hierarchies should be based on interface classes.*

### Rationale

An interface class has these properties:

- If the class has member functions, they are public and pure `virtual`.
- If the class has data members, they are public and `static constexpr`.

Using an interface class as a base class in a hierarchy:

- Separates the interface and implementation. The code of the base class is more stable and easier to maintain.
- Avoids unnecessary computations of nonstatic data members in the derived classes and other compilation dependencies.
- Makes your software easier to extend and enables the use of alternative implementations through the same interface.

### Polyspace Implementation

Polyspace flags the base of a class hierarchy if that base class is not an interface.

When class definitions are nested in other classes, the checker follows these conventions:

- Non-interface base classes are flagged even if the hierarchy is nested inside another class. For example, in this code snippet, class `NestedBase` is flagged :

  ```
  class ClassWithNestedHierarchy
  {
      class NestedBase //Non-compliant, not an interface
      {
      public:
          int i;
      };

      class NestedDerived : public NestedBase
      {
      public:
          int j;
      };
  };
  ```
- Base classes with nested non-interface classes are not flagged. For example, in this code snippet, `NestedClass` is not an interface class but the outer class `InterfaceWithInnerClass` is not flagged when used as a base class:

```
   class InterfaceWithInnerClass
   {
   public:
       class NestedClass   //not an interface class
       {
       private:
           int i;
       };

       static constexpr NestedClass i{};
   };

   class DerivedBaseWithInnerClass : public InterfaceWithInnerClass
   {
   private:
       int i;
   };
```

#### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

#### Interfaces as Base Classes in Hierarchies

```
class Interface
{
public:
    virtual ~Interface() = 0;
    virtual void SomeFunc() = 0;
};

class NotAnInterface //Non-compliant
{
public:
    void Implementation() {}

};

class IsDerived1 : public Interface
{

public:
    ~IsDerived1() {}

    void SomeFunc() final {}

};


class IsDerived2 : public NotAnInterface
{
public:
    IsDerived2() = default;
};


template <typename T>
class TmplInterface
{
public:
    virtual T func() noexcept = 0;

};
template<typename T>
class TmplNotInterface //Non-compliant
{
public:
    T func2();
};

template <typename T>
class TmplDerived: public TmplInterface<T>, TmplNotInterface<T>
{
public:
    T func() noexcept override { return t;}

    T t;
};

class TmplDerived<int> var;
```

In this example, the checker flags:

- The non-interface class `NotAnInterface`, which acts as a base for the class `IsDerived2`. The class `NotAnInterface` is not an interface class because it contains a nonvirtual member function.
- The template non-interface class `TmplNotInterface`, which acts as a base for the class `TmplDerived`. Note that `TmplDerived` also derives from the interface class `TmplInterface`, which is compliant with the rule.

## Check Information

**Group:** Derived Classes
**Category:** Advisory, Non-automated

# Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A11-0-1

A non-POD type should be defined as class

## Description

### Rule Definition

*A non-POD type should be defined as class.*

### Rationale

A POD (Plain Old Data) type can be exchanged with C code in its binary form, and can be safely copied by using the `std::memcpy` function. Scalar types, C-style structures and unions, and arrays of these types are all examples of POD types. However, the C++ language also allows you to create structures and unions that are non-POD types. Such structures and unions can provide custom-defined constructors, have nonstatic data members with private or protected access control, have an interface, and implement an invariant.

A software developer typically expects object-oriented concepts such as encapsulation to be implemented by using classes. In addition, a class specifier forces the type to provide private access control for all its members by default and is naturally suited for implementing encapsulated types. So, to create easily readable and maintainable code, define a non-POD type as a class instead of a structure or a union.

### Polyspace Implementation

The checker flags a structure or a union in your code is not a POD type. This includes structures and unions that are instantiated by using templates.

For a simplified explanation of a POD type in C++ language, see the previous section. For a full specification of a POD type, see the C++ reference manual.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Define a Non-POD Type as a Class

```
#include <cstdint>
#include <limits>

class A // Compliant, non-POD type defined as class
{
    std::int32_t x; // Data member is private by default

  public:
    static constexpr std::int32_t maxValue = std::numeric_limits<std::int32_t>::max();
    A() : x(maxValue) {}
    explicit A(std::int32_t number) : x(number) {}
    std::int32_t GetX() const noexcept
    {
      return x;
    }

    void SetX(std::int32_t number) noexcept
    {
      x = number;
    }
};

struct B // Noncompliant, non-POD type defined as struct
{
  private:
    std::int32_t x;  // Must provide private access specifier for x member

  public:
    static constexpr std::int32_t maxValue = std::numeric_limits<std::int32_t>::max();
    B() : x(maxValue) {}
    explicit B(std::int32_t number) : x(number) {}

    std::int32_t GetX() const noexcept
    {
      return x;
    }

    void SetX(std::int32_t number) noexcept
    {
      x = number;
    }
};
```

Both `class A` and `struct B` implement the same non-POD type. This type has:

- A nonstatic data member `x` that has private access control.
- Two user-defined constructors. The default constructor initializes `x` to the maximum value that an `int32` type can store. The constructor that has one parameter disallows implicit conversion.
- An interface provided by the `GetX` and the `SetX` methods to access and modify the state of an object.

It is preferable that you implement this type, that encapsulates its contents, as a class.

The definition of `class A` complies with this coding rule. The definition of `struct B` violates this coding rule.

## Check Information

**Group:** Member access control
**Category:** Advisory, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A11-0-2

A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class

## Description

### Rule Definition

*A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.*

### Rationale

The items prohibited by this rule are not supported for `struct` types in C code. In C++, a `struct` type can have private data members, member functions, be inherited and inherit from other `struct`-s or `class`-es. However, a developer typically associates these features with a `class` type. Adhering to this rule makes sure that you use only classes to implement object oriented concepts such as data encapsulation and inheritance.

Adhering to this rule also makes sure that your `struct` types conform to the rules of Plain Old Data (POD) types and can be exchanged with C code.

### Polyspace Implementation

The checker flags `struct` types with one or more of these features:

- Contains private or protected data members.

  `struct` members are public by default.
- Contains member functions.
- Acts as base class for another `struct` or `class`, or inherits from another `struct` or `class`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Struct Types with Class-Like Features

```
#include <cstdint>
#include <iostream>

struct loginCredentials1 { //Noncompliant: Private members
    int32_t username;
private:
    int32_t pwd;
};
```

```
struct loginCredentials2 {   //Noncompliant: Member functions
    int32_t username;
    int32_t pwd;
    void readFromFile(std::string fileName) {
        //Read members data from file
    }
};

struct loginCredentials3 {  //Noncompliant: Acts as base for another struct
    int32_t username;
    int32_t pwd;
};

struct adminLoginCredentials: loginCredentials3 { //Noncompliant: Inherits from another struct
    std::string permissions;
};
```

In this example, all `struct` types are noncompliant.

- `loginCredentials1` contains a private data member `pwd`.
- `loginCredentials2` contains a member function `readFromFile()`.
- `loginCredentials3` acts as a base for the struct `adminLoginCredentials`.
- `adminLoginCredentials` inherits from the struct `loginCredentials3`.

## Check Information

**Group:** Member access control
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A11-3-1

Friend declarations shall not be used

## Description

### Rule Definition

*Friend declarations shall not be used.*

### Rationale

You declare a function as friend of a class to access private members of the class outside the class scope.

```
class A
{
  int data;
  public:
    // operator+ can access private members of class A such as data
    friend A const operator+(A const& lhs, A const& rhs);
};
```

Friend functions and friend classes reduce data encapsulation. Private members of a class are no longer accessible only through the class methods.

Code with friend functions can be difficult to maintain. For instance, if class `myClass` has a friend class `anotherClass`, when you change a data member of `myClass`, you have to find all instances of its usage in member functions of `anotherClass`.

### Polyspace Implementation

The rule checker flags all uses of the `friend` keyword.

The checker follows specifications of AUTOSAR C++ 14 release 18-03 (March 2018). However, release 18-10 and later releases of AUTOSAR C++14 allows an exception for comparison operators such as `operator==`. If the rule checker flags the use of comparison operators, add a comment justifying the result. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `friend` Keyword

```
class myClass
{
    int data;
public:
    myClass& operator+=(myClass const& oth);
    friend myClass const operator+(myClass const& lhs, // Noncompliant: Use of friend keyword
                                   myClass const& rhs);

};
```

`operator+` is a friend function of class `myClass` and can access its private member, `data`. The presence of this friend function violates the rule.

## Check Information

**Group:** Member Access Control
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-0-1

If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well

## Description

### Rule Definition

*If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.*

### Rationale

These special member functions are called for copy or move operations:

- Copy constructor
- Copy assignment operator
- Move constructor
- Move assignment operator
- Destructor

If you do not explicitly declare any of these functions, the compiler defines them implicitly. This implicit definition implements shallow copying of objects and can cause errors. If you need to explicitly declare any of the special member functions, you must declare all of them. For instance, suppose you want to copy an object that contains a raw pointer to a dynamically allocated memory. The implicit copy constructor shallow-copies the object, after which the original pointer and the copied pointer point to the same memory. If one of the objects is destroyed, the allocated memory is deallocated, leaving a dangling pointer in the other object. Accessing the dangling pointer can cause segmentation errors. Because all the special member functions are closely related, the implicit implementation of the other functions can lead to similar errors. To manage the life cycle of the dynamically allocated resource, explicitly declare all five of the special member functions (Rule of Five). Alternatively, you can use objects where memory management is correctly implemented in the implicit definition of the special member functions and explicitly declare none of them (Rule of Zero).

When you explicitly declare some but not all of the special member functions, the compiler can prevent the use of the undeclared special member functions. For example, if you explicitly declare only the copy constructor or destructor functions of a class, the compiler no longer defines the move constructor and move assignment operator implicitly. The class becomes a copy-only class, perhaps inadvertently. Conversely, if you explicitly declare only the move constructor and move assignment operator, the compiler disables the copy constructor and copy assignment operator by defining them as deleted. The class becomes a move-only class, which might not have been your intention. To avoid such unwanted effects, either follow the Rule of Five or follow the Rule of Zero.

The constructor of a class is not part of this rule.

### Polyspace Implementation

Polyspace flags classes that explicitly declare some but not all of the five special member functions. Note that the move constructor and move assignment operators were introduced in C++11. Polyspace does not make any exception for older codes.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Follow Either The Rule of Five or The Rule of Zero

This example demonstrates the Polyspace implementation of AUTOSAR rule A12-0-1.

```
// Class rendered copy-only, perhaps inadvertently
class A               // Noncompliant.
{
    public:
      ~A()
      {
        // ...
      }

    private:
      // Member data ...
};


//Class rendered move-only, perhaps inadvertently
class B          // Noncompliant
{
   public:
        B(B&&) = default;
        B& operator=(B&&) = default;
    private:
      // Member data ...

};
template<typename T>
class BaseT              // Compliant - rule of five.
{
  public:
    BaseT(BaseT const&) = delete;
    BaseT(BaseT&&) = delete;
    virtual ~BaseT() = default;
    BaseT& operator=(BaseT const&) = delete;
    BaseT& operator=(BaseT&&) = delete;
    protected:
    BaseT() = default;
};

template<typename T>
class SimpleT           // Compliant - rule of zero.
{
  public:
    SimpleT(T t): t_(t)
    {

    }

  private:
    T t_;
};

main()
{
    //..
}
```

The class `A` declares only its destructor, which makes this class copy-only because the compiler no longer defines the move constructor and move assignment operator. The class `B` declares the move constructor and the move assignment operator, which makes this class move-only because the compiler disables the copy constructors and copy assignment operators. It is not clear whether these effects are deliberate. Polyspace flags these declarations and indicates which special member functions are missing. The class `BaseT` is compliant with this rule because all five of the special member functions are declared. Similarly, `SimpleT` is compliant because it declares none of the special member functions and relies on their implicit definition.

### Check Information

**Group:** Special member functions
**Category:** Required, Automated

## Version History

**Introduced in R2020a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-0-2

Bitwise operations and operations that assume data representation in memory shall not be performed on objects

## Description

### Rule Definition

*Bitwise operations and operations that assume data representation in memory shall not be performed on objects.*

### Rationale

In C++, object representation in memory might include:

- Data members declared with different access privileges
- Bit-field data members
- Padding bytes between data members
- Padding bytes at the end of data members
- Pointers to the vtable to support virtual functions

The arrangement of these different parts of an object in memory is environment dependent. In addition, static data members or function members of an object are stored in a separate physical location in memory. When you perform bitwise operation on an object by assuming certain arrangement of data in memory, you might inadvertently assume incorrectly, and access bits are not part of the value representation of the object. Accessing these bits can lead to undefined behavior.

Consider this class that contains a virtual function:

```
class notPOD{
public:
    virtual void foo();
    int value;
protected:
    double dvalue;

};
//...
int main(){
    notPOD Obj;
    std::memset(&Obj, 57, 2); // attempts to set Obj::value to 57
}
```

When `Obj` is stored in a memory block, the block contains a pointer to the virtual table in addition to the variables `Obj::value` and `Obj::dvalue`. The size of this pointer or its location in memory can depend on the environment. In `main()`, `std::memset()` attempts to set the value of `Obj::value` by assuming that:

- `Obj::value` is the first block in the memory representation of `Obj`.
- `Obj::value` is represented by 2 bytes in the memory.

Because these assumptions are generally not correct, using `std::memset()` can lead to undefined behavior. For instance, if you inadvertently modify the pointer to virtual table, calling `foo()` can invoke an unexpected function.

The representation of class and structures in memory is environment-dependent and can contain additional bytes alongside the value representation. Relying on the data representation of an object to perform bitwise operations can result in modifying bits that are not part of the value representation, leading to undefined behavior. Avoid operations that assume a certain representation of an object in memory to access its bits. To perform operations on a class, use dedicated member functions, overloaded operators, or mutators.

### Polyspace Implementation

The C functions that access accesses memory bits includes `std::memset()`, `std::memcpy()`, `std::memmove()`, `std::strcpy()`, `std::memcmp()`, `std::strcmp()`. Polyspace flags a statement when:

- You use the C functions to initialize or copy initialize nontrivial objects
- You use the C functions to compare nonstandard layout objects
- You use the C functions on any objects that contain padding data

The statements containing the noncompliant operations are flagged and relevant class declarations are highlighted. For definitions of trivial and standard layout classes, see the C++ Standard, [class], paragraphs 6 and 7 respectively.

As an exception, Polyspace does not flag operations that use the C functions to access the memory bits of trivial and standard layout objects with no padding data. Although using bitwise operation on trivial and standard layout classes with no padding data complies with this rule, it is not a good practice. Instead, use dedicated member function, overloaded operators, or mutators.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Bitwise Operations on Objects

Consider this code that contains these classes:

- `TrivialClass` is a trivial class with padding data and an overloaded `operator|=`.
- `NonTrivialClass` is a nontrivial class with a virtual function and an overloaded `operator==`.

These classes are represented in different ways in the memory. This example shows how Polyspace flags bitwise operations that are performed on such objects.

```
#include <cstdint>
#include <cstring>
class TrivialClass
{
public:
    TrivialClass() = default;
    TrivialClass(uint8_t c, uint32_t i, int8_t d) :
```

```
    c(c), i(i), d(d) {}
    TrivialClass& operator |=(const TrivialClass& other)
    {
        uint32_t buf[4] {this->c|other.c,this->i|other.i,this->d|other.d};
        memcpy(this, buf, sizeof(uint32_t) * 3); //Noncompliant
        return *this;
    }

private:
    uint8_t c;
    uint32_t i;
    int8_t d;
};

class NonTrivialClass
{
public:
    NonTrivialClass() = default;
    NonTrivialClass(uint32_t a, uint32_t b, uint32_t c) :
    a(a), b(b), c(c){}
    bool operator==(const NonTrivialClass& rhs) const noexcept
    {
        return a==rhs.a && b==rhs.b && c==rhs.c;
    }
    virtual ~NonTrivialClass() {}
private:
    uint32_t a;
    uint32_t b;
    uint32_t c;
};

int main(void)
{
    TrivialClass A, A1{3,5,7};
    NonTrivialClass B, B1{10,11,12};
    std::memset(&A, 3, 1); //Noncompliant
    A |= A1;
    if (!std::memcmp(&A, &A1, sizeof(TrivialClass))) {} //Noncompliant
    std::memcpy(&B, &B1, sizeof(NonTrivialClass)); //Noncompliant
    if (B == B1){} //Compliant
    return 0;
}
```

- Polyspace flags the statement `std::memset(&A, 3, 1);` because in this statement, `std::memset()` modifies the individual bits in the memory representation of the trivial object `A` including padding data. Accessing padding data bits of an object is a violation of this rule even if the object is a trivial class object. For the same reason, Polyspace flags the statement in the definition of `TrivialClass::operator|=` containing `memcopy()`.
- Polyspace flags the statement `std::memcpy(&B, &B1, sizeof(NonTrivialClass));` because `std::memcpy()` accesses the individual bits in the memory representation of the nontrivial object `B` including the pointer to vtable. This pointer is not part of the value representation and accessing this pointer is a violation of this rule.
- Polyspace does not flag the statement `if(B==B1)` because `NonTrivialClass` has an overloaded `operator==` that can compare `B` and `B1` without accessing their individual bits.

## Check Information

**Group:** Special member functions
**Category:** Required, Partially automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A12-1-1

Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members

## Description

### Rule Definition

*Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members.*

### Rationale

If a derived class does not explicitly initialize the constructors of its base class, then the compiler implicitly initializes the base class. An object of invalid state or an object with unintended initial values might be constructed, which risks unexpected code behavior during run time. Consider this diamond class hierarchy, where the base class `Parent` has multiple constructors.

```
class Parent{
    public:
    Parent(){/*...*/}
    Parent(int i){/*...*/}
};

class Child1: public virtual Parent{
    public:
    Child1(): Parent(2){/*...*/}
};

class Child2: public virtual Parent{
    public:
    Child2(): Parent(1){/*...*/}

};

class GrandChild: public Child1, public child2{
    public:
    GrandChild(){/*...*/}
}
```

When constructing a `GrandChild` object, it is unclear whether `Parent` is constructed by using `1` or `2` as the argument. Having `GrandChild` explicitly specify the constructor used to initialize the `Parent` resolves the ambiguity. To avoid invalid state and unintended initial values, directly call the necessary base class constructors in the derived class constructor initialization list.

### Polyspace Implementation

Polyspace flags the constructor of a derived class if its initialization list:

- Does not explicitly call the constructors of the virtual base classes.
- Does not explicitly call the constructors of the direct nonvirtual base classes.
- Does not explicitly initialize nonstatic data members.

If a nonstatic data member is initialized in at least one branch of a branching, looping, or exception handling statement, Polyspace considers the data member to be initialized.

Polyspace does not report a violation of this rule if you use delegating constructors.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Class Construction When Using Multiple Inheritance

```
#include <cstdint>

class A {
public:
    A() : a{1} {}
    virtual void abstractA() const = 0;
private:
    int a;
};

class B : public  A {
public:
    B() : b{1} {} //Noncompliant
    void abstractA() const override {}
private:
    int b;
};

class C {
public:
    C() : c{3} {}
private:
    int c;
};

class D : public B, public C {
public:
    D() : B(), C(), e{5} {} //Compliant
private:
    int e;
};

int main() {
    D dName;
    return 0;
}
```

In this example, Polyspace flags the class constructors that do not explicitly initialize the base classes and nonstatic data members. For instance:

- The B class constructor is noncompliant because it is missing an explicit initialization of its base class A. To resolve this issue, call the A class constructor in the B constructor initialization list.

- The D class constructor is compliant because it explicitly initializes both of its direct nonvirtual base classes through initialization list constructor calls.

### Class Construction When Using Multiple and Virtual Inheritance

```
#include <cstdint>
class A {
public:
    A() : a{1} {}
    virtual void abstractA() const = 0;
private:
    int a;
};


class B : public virtual A {
public:
    B() : A(), b{1} {} //Compliant
    void abstractA() const override {}
private:
    int b;

};


class C : public virtual A {
public:
    C() : c{3} {} //Noncompliant
    void abstractA() const override {}
private:
    int c;
};


class D : public B, public C {
public:

    D() : B(), C(), e{5} {} //Noncompliant
private:
    int e;
};



int main() {
    D dName;
    return 0;
}
```

In this example, Polyspace flags the class constructors that do not explicitly initialize the base classes and nonstatic data members. For instance:

- The B class constructor is compliant because it explicitly initializes its direct base class within its initialization list.
- The C class constructor is noncompliant because it does not explicitly call its direct base class A. To resolve this issue, call the A class constructor in the C constructor initialization list.

- The D class constructor is noncompliant because it does not explicitly call its virtual base class A. As a result of multiple and virtual inheritance, the most derived class must initialize the virtual base class. To resolve this issue, call the A class constructor in the D constructor initialization list.

### Class Construction by Using Delegating Constructor

In C++11 and later, if a class contains more than one constructors, the class design can be simplified by using delegating constructor. When you use delegating constructors, it is not necessary to initialize all data members and base classes in each constructor. In this example, Polyspace does not report violations of this rule when the constructor `Derived::Derived(int)` delegates its initialization duties to `Derived::Derived(int, int)`.

```
class Base {
public:
    Base() : a{1} {}

    virtual void abstractA() const = 0;

private:
    int a;
};
class Derived: public Base {
public:
    Derived(int a, int b): Base{}, z{a}, y{b} {}  //Compliant
    Derived(int a): Derived(a, 0) {}  //Compliant

private:
    int z;
    int y;
};
```

## Check Information

**Group:** Special Member Functions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A12-1-2

Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type

## Description

### Rule Definition

*Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.*

### Rationale

You can initialize a non-static data member of a class in one of these ways:

- In the declaration of the member in the class body by using the non-static data member initializer (NSDMI)
- By using a non-static member initializer in a constructor

In a class, initializing a subset of the non-static data members by using the NSDMI and initializing the remaining non-static data members by using a constructor reduces code readability. This code pattern might cause confusion for the reader about which initial values for each data member are actually used. Using either the NSDMI or a constructor to initialize all non-static data members of the class avoids this potential confusion.

The move and copy constructors are exempt from this rule because these constructors have the special behavior of initializing the data members by using their existing values from other objects. These constructors are unlikely to cause the confusion for the reader.

### Polyspace Implementation

If you use the NSDMI to initialize a subset of the non-static data members of a class and a constructor to initialize the remaining non-static data members, the checker flags the constructor and the associated NSDMI initializations.

The checker does not flag the move and copy constructors that violate this rule.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Using Both NSDMI and Constructor Initializer

```
#include <cstdint>
#include <utility>

class A
{
  public:
    A() : i1{0}, i2{0} // Compliant, i1 and i2 are initialized by the constructor only
    {
    }

  private:
    std::int32_t i1;
    std::int32_t i2;
};

class B
{
  private:
    std::int32_t i1{0};
    std::int32_t i2{0}; // Compliant, i1 and i2 are initialized by NSDMI only
};

class C
{
  public:
    C() : i2{0}   // Noncompliant, i1 is initialized by NSDMI, i2 is initialized by constructor
    {
    }

  private:
    std::int32_t i1{0};
    std::int32_t i2;
};

class D
{
    D(D const& oth) : i1{oth.i1}, i2{oth.i2} // Compliant by exception, copy constructor
    {
    }

    D(D&& oth): i1{std::move(oth.i1)}, i2{std::move(oth.i2)}
    // Compliant by exception, move constructor
    {
    }

  private:
    std::int32_t i1{0};
    std::int32_t i2{0};
};
```

In this code, only the constructor in class C does not comply with this coding rule because:

- The data member i1 is initialized by using the NSDMI.
- The data member i2 is initialized in the constructor.

## Check Information

**Group:** Special member functions
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-1-3

If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead

## Description

### Rule Definition

*If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead.*

### Rationale

You might be using user-defined class constructors to initialize the nonstatic data members of your class. When all user-defined constructors initialize nonstatic data members to the same value, using a user-defined constructor for initialization purposes is unnecessary. The best practice is to initialize the nonstatic data members directly in the class definition. Such in-class nonstatic data member initialization (NSDMI) enables you to use the error-free and efficient implicit constructor to obtain an instance of the class that has the data members initialized to a default values.

### Polyspace Implementation

Polyspace flags a nonstatic data member declaration if either of these conditions is true:

- The nonstatic data member is not initialized in-class and all user-defined constructors initialize the data member to the same value.
- The nonstatic data member is initialized in-class and at the same time, it is also initialized in user-defined constructors.

This checker does not apply to:

- Copy and move constructors
- Union definitions
- Arrays that are initialized in constructors
- Objects that are initialized field by field in constructors

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Initialize Nonstatic Data Members With Default Values

```
#include <cstdint>
#include <string>
class MessageBox1
{
```

```
public:
    MessageBox1() : LowerLeft(0), UpperRight(0.0F), msg()
    {
    }
    // ...

private:

    int LowerLeft;     // Noncompliant
    float UpperRight;  // Noncompliant
    std::string msg;    // Noncompliant
};

class MessageBox2
{
public:
    // ...

private:
    int LowerLeft = 0;     // Compliant
    float UpperRight = 0.0F;         // Compliant
    std::string msg = "";   // Compliant
};
class MessageBox3
{
public:
    MessageBox3() : LowerLeft(0), UpperRight(0.0F), msg()
    {
    }


private:
    int LowerLeft = 0;      // Noncompliant
    float UpperRight = 0.0F;          // Noncompliant
    std::string msg = "";    // Noncompliant
};
class MessageBox4
{
public:
    MessageBox4() : LowerLeft(0), UpperRight(0.0F), msg()
    {
    }
    // ...
    MessageBox4(int int_i): LowerLeft(int_i),UpperRight(0.1F), msg("str"){}
private:

    int LowerLeft;     // Compliant
    float UpperRight;            // Compliant - Initialized differently in two c'tor
    std::string msg = "";//Noncompliant -  Initialized differently in two c'tor
};
```

In this example, Polyspace flags nonstatic data member initializations that violate this rule. For instance:

- In class `MessageBox1`, the declarations of the data members are noncompliant because they are initialized by a user-defined constructor instead of by in-class initialization. The best practice is to declare such data members directly in-class and to use the default implicit constructors.

- In class `MessageBox2`, the declarations of the data members are compliant because they are initialized directly in-class and the class defines no user-defined constructor.
- In class `MessageBox3`, the declarations of the data members are noncompliant because they are initialized in-class while the constructor of the class also initialize the data members. The best practice is to declare the data members in class and omit a user defined constructor.
- In class `MessageBox4`:

  - The declaration of `AA::LowerLeft` and `AA::UpperLeft` are compliant because two different constructors initialize them to different values and they are not initialized in-class.
  - The declaration of `AA::msg` is noncompliant because two different constructors initialize it to different value and it is also initialized in-class.

### Check Information

**Group:** Special member functions
**Category:** Required, Automated

## Version History

**Introduced in R2021b**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-1-4

All constructors that are callable with a single argument of fundamental type shall be declared explicit

## Description

### Rule Definition

*All constructors that are callable with a single argument of fundamental type shall be declared explicit.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Special Member Functions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A12-1-5

Common class initialization for non-constant members shall be done by a delegating constructor

## Description

### Rule Definition

*Common class initialization for non-constant members shall be done by a delegating constructor.*

### Rationale

C++ classes often have several constructors with different syntaxes. These initializers might have some initializations in common. For instance, in this code, both constructors of the class `Circle` initialize the nonconstant members `x`, `y` and `r`.

```
class Circle{
    int x;
    int y;
    int r;
    public:
    Circle(int x_in, int y_in, int r_in): x{x_in}, y{y_in},r{r_in}{
        //...
    }
    Circle( int x_in, int y_in): x{x_in}, y{y_in}, r{0}{
        //...
    }
    //...
};
```

It is expected that all constructors of a class have equivalent behavior. When these common tasks are performed repeatedly in multiple constructors, any inadvertent differences might lead to confusion and unexpected results. Performing the common tasks repeatedly can also be tedious.

To avoid unexpected results, delegate the initialization of nonconstant members to existing constructors whenever possible. Avoid repeating initializations in multiple constructors.

### Polyspace Implementation

Polyspace flags a class, union, or structure if any of their nonconstant members are initialized in multiple constructors. Polyspace does not flag:

- Copy or move constructors that do not use delegate constructors.
- Arrays that are initialized in multiple constructors.
- Objects that are initialized field by field in multiple constructors.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Repeating Initialization in Multiple Constructors

```
#include <cstdint>
#include <string>
#include<initializer_list>

class Circle // Noncompliant
{
public:
    Circle(std::int32_t xx, std::int32_t yy, std::int32_t rr):x{xx}, y{yy}, r{rr}
    {}

    Circle(std::int32_t xx, std::int32_t yy):Circle(xx,yy,0) //Delegated constructor
    {}

    Circle(std::int32_t xx):x(xx),y(1),r(1) // Could be delegated
    {}

    Circle():x(0),y(0),r(0)//Could be delegated
    {}

protected:
    std::int32_t x;
    std::int32_t y;
    std::int32_t r;
};
class Flag // Compliant
{
public:
    Flag(bool a):a(a)
    {}

    Flag():a(0)
    {}

protected:
    const bool a;
};

class Tuple    // Noncompliant
{
public:
    Tuple(std::initializer_list<float> ilist) {
        auto* p = ilist.begin();
        x = *p++;
        y = *p++;
        z = *p++;
    }
    Tuple(float _x, float _y, float _z) : x(_x), y(_y), z(_z) {}

    float x = 0, y = 0, z = 0;
};
```

This example shows compliant and noncompliant classes.

- The class `Circle` has four constructors. It delegates the common initializations in the second constructor, but repeats the initializations in the third and fourth constructors. This repetition might lead to unexpected results. Polyspace flags the class.
- The class `Flag` repeats the initialization of its member `a` in two constructors. Because `a` is a `const`, Polyspace does not flag the class.
- The class `Tuple` initializes its non`const` members repeatedly in two constructors. Polyspace flags the class.

## Check Information

**Group:** Special member functions
**Category:** Required, Partially automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-1-6

Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors

## Description

### Rule Definition

*Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.*

### Rationale

If a derived class uses all the base class constructors and does not explicitly initialize any additional data member that is not in the base class, reimplementing these constructors in the derived class adds unnecessary lines of code. The reimplementation might not exactly match the constructors in the base class due to human error and might introduce bugs in your code. Moreover, every time you change the base class constructors, you must also update the derived class constructors. This increases the overhead of code maintenance.

In such situations, using inheriting constructors in the derived class solves these issues.

### Polyspace Implementation

Polyspace flags a class for violation of this rule if the class satisfies all of these conditions:

- The class derives from a single base class.
- The class uses all the base class constructors and reimplements them in the class definition.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Derived Class Reimplements the Constructors

```
#include <cstdint>

class A
{
  public:
    A(std::int32_t x, std::int32_t y) : x(x + 8), y(y) {}
    explicit A(std::int32_t x) : A(x, 0) {}

  private:
    std::int32_t x;
    std::int32_t y;
};

class B : public A    // Non-compliant
```

```
{
  public:
    B(std::int32_t x, std::int32_t y) : A(x, y) {}
    explicit B(std::int32_t x) : A(x) {}
};

class C : public A   // Compliant
{
  public:
    using A::A;
};
```

The reimplementation of `B(std::int32_t x, std::int32_t y)` and `explicit B(std::int32_t x)` violates the rule because they are identical to the base class constructors `A(std::int32_t x, std::int32_t y)` and `explicit A(std::int32_t x)`.

## Check Information

**Group:** Special member functions
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-4-1

Destructor of a base class shall be public virtual, public override or protected non-virtual

## Description

### Rule Definition

*Destructor of a base class shall be public virtual, public override or protected non-virtual.*

### Rationale

If a base class destructor is not public virtual or public override, the class cannot behave polymorphically for deletion of derived class objects.

If a pointer to a base class refers to a derived class object and you use the pointer to delete the object:

```
class Base {
  public:
    ~Base() {}
};

class Derived: public Base {
  public:
    ~Derived() {}
};
...
void func(Base* ptr) {
    //ptr might point to a Base or Derived object
    delete ptr;
}
```

only the base class destructor is called. Additional resources allocated in the derived class are not released and can cause a resource leak. See example below.

If you want to prevent calling the derived class destructor through a base class pointer, make your intent explicit by making the destructor protected. Otherwise, it might appear that the possibility of polymorphic deletion of derived class objects was not considered.

### Polyspace Implementation

The checker flags base classes with destructors that are not public virtual, public override or protected non-virtual.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Base Class Destructor Not Virtual

```
#include <new>

class Base {
  public:
    Base() {}
    ~Base() {} //Noncompliant
};

class Derived: public Base {
     int *arr;
  public:
     Derived() {
        arr = new int(5);
     }
    ~Derived() {
        delete arr;
     }
};

void main() {
    Base* basePtr = new Derived();
    delete basePtr;
}
```

In this example, the class `Base` has a non-virtual destructor. As a result, when the pointer `basePtr` is deleted, only the destructor of class `Base` is invoked. However, `basePtr` points to an object of class `Derived`. The deletion is not complete because the destructor of class `Derived` is not invoked. In particular, the data member `arr` in the derived object is not deleted.

## Check Information

**Group:** Special member functions
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-4-2

If a public destructor of a class is non-virtual, then the class should be declared final

## Description

### Rule Definition

*If a public destructor of a class is non-virtual, then the class should be declared final.*

### Rationale

In C++, when any object of a derived class is destroyed, first the destructor of its class is invoked, and then the destructors of the base classes are invoked. Class hierarchies can also be polymorphic. You can declare a base class pointer and assign a derived class object to it. To safely destroy objects belonging to a class hierarchy, declare the `public` class destructors as `virtual`. Consider this code where two base class pointers that point to derived objects are destroyed.

```
class Base{
public:
    virtual    ~Base();
    //..
};

class Derived : public Base{
public:
    ~Derived();
    //..
};
class Base2{
public:
    ~Base2();
    //..
};

class Derived2 : public Base2{
public:
    ~Derived2();
    //...
};
int main(){

    Base* ptr = new Derived;
    Base2* ptr2 = new Derived2;
    delete ptr;
    delete ptr2;
}
```

- The object `ptr` is a pointer of class `Base` that points to an object of class `Derived`. When `ptr` is deleted, the destructor of the derived class is called first, and then the destructor of the base class is called. Even though `ptr` is a base class object, the correct destructors are called to release all acquired resources because the `public` destructors in this class hierarchy are declared as `virtual`.

- When the pointer ptr2 is deleted, the destructor of only the base class is called because the public destructors in this class hierarchy are nonvirtual. This kind of incomplete destruction is undefined behavior, which can lead to memory leaks and unexpected termination of code execution.

To prevent undefined behavior, do not use classes with public nonvirtual destructors as base classes. Declare such classes as final to specify that these classes are not base classes and new classes cannot be derived from them.

### Polyspace Implementation

Polyspace flags a class declaration if both these statements are true:

- The public destructor of the class is not declared as virtual.
- The class is not declared final.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Declare a Class as final if Its Public Destructor Is Nonvirtual

This example shows how Polyspace flags base classes that have public nonvirtual destructors.

```
#include<cstdint>
class Base{    //Noncompliant
public:
    ~Base();
    //..
};

class Derived : public Base{  //Noncompliant
public:
    ~Derived();
    //..
};
class Base2 final{  //Compliant
public:
    ~Base2();
    //..
};

//class Derived2 : public Base2{ //Compilation error
//public:
//    ~Derived2();
//    //...
//};
int main(){

    Base* ptr = new Derived;
//    Base2* ptr2 = new Derived2;  //Compilation Error
    delete ptr;
    //    delete ptr2;
}
```

The classes `Base` and `Derived` have public nonvirtual destructors. In `main()`, when `ptr` is destroyed, only `~Base()` is called, resulting in partial destruction of the pointed-to object. This behavior is undefined behavior that can lead to memory leak and unexpected program termination. Polyspace flags the declaration of both `Base` and `Derived`.

The class `Base2` has a public nonvirtual destructor. `Base2` is compliant with this rule because it is declared as `final`. Deriving any class from `Base2` results in compilation failure. Consequently, you cannot declare a pointer of class `Base2` that points to an object of a derived class. Declaring classes with public nonvirtual destructors as `final` prevents undefined behaviors and can protect the code from memory leaks and unexpected program termination.

### Declare Nonvirtual Destructors as `protected`

This example shows that Polyspace allows nonvirtual destructors when they are declared `protected`.

```
#include<cstdint>
class Base{    //Compliant
protected:
    ~Base();
    //..
};

class Derived : public Base{  //Compliant
protected:
    ~Derived();
    //..
};

int main(){

    Base* ptr = new Derived;
    delete ptr;//Compilation error
}
```

Nonvirtual destructors declared as `protected` are compliant with this rule. Because the destructor for `Base` is protected, the statement `delete ptr;` causes a compilation failure. Declaring nonvirtual destructors as `protected` can prevent memory leaks and unexpected program termination. When nonfinal classes have nonvirtual destructors declared as `protected`, the classes comply with this rule and Polyspace does not flag them.

## Check Information

**Group:** Special member functions
**Category:** Advisory, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-6-1

All class data members that are initialized by the constructor shall be initialized using member initializers

## Description

### Rule Definition

*All class data members that are initialized by the constructor shall be initialized using member initializers.*

### Rationale

It is inefficient to initialize data members of a class by assigning a copy of passed values to them in the body of a constructor. For instance, this code is inefficient:

```
class foo{

private:
    int i;
public:
    foo(int input){
        i = input;
        //...
    }
};
```

It is more efficient to initialize data members of classes by using member initializers. For instance:

- Initialize data members by using a initializer list.
- Initialize data members by using default member initializers.

To increase the efficiency of your code and to protect your code from using an uninitialized data member, use the preceding methods to initialize data members of a class.

### Polyspace Implementation

Polyspace flags the constructor definition of a class if the constructor initializes the nonstatic data members of the class in its body by copying the passed values to the data members. Polyspace does not flag constructors with uninitialized static data members.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Efficient Initialization of Class Data Members

This example shows efficient initialization methods of class data members that are compliant with this rule.

```
#include <cstdint>
#include <string>
using namespace std;
class A
{
public:
    A(int32_t n, string s) : number{n}, str{s}
    { //Compliant
        n += 1; // This does not violate the rule
        str.erase(str.begin(),
        str.begin() + 1); // This does not violate the rule
        // Implementation
    }

private:
    int32_t number;
    string str;
};

class C
{
public:
    C(int32_t n, string s)
    { //Compliant
        n += 1; // This does not violate the rule
        str.erase(str.begin(),
        str.begin() + 1); // This does not violate the rule
    }
    // Implementation

private:

    int32_t number = 0;
    string str = "string";
    static double pi;
};
```

- The constructor of class A initializes the data members by using an initializer list. This constructor is compliant with this rule.
- The constructor of class C initializes the data members by using default initialization. These data members cannot be used before they are initialized. This constructor is compliant with this rule. Polyspace does not flag constructors that do not initialize static data members.

### Inefficient Initialization of Class Data Members

This example shows inefficient initialization of class data members that is not compliant with this rule.

```
#include <cstdint>
#include <string>
using namespace std;
class B
{
public:
    B(int32_t n, string s)
    { //Noncompliant
        number = n;
        str = s;
    }
    // Implementation

private:
    int32_t number;
    string str;
};
class E{
public:
    E():E(1,"string")
    {

    }
    E(int32_t a, string str) : number(a)
    {//Noncompliant

    }
private:
    int32_t number;
    string str;
};
```

- The constructor of class B initializes the data members by copying the passed parameters. This initialization is inefficient. The data members of class B might be used before they are initialized. Polyspace flags this inefficient and risky constructor.
- The default constructor of class E attempts to initialize the nonstatic data members by delegating the initialization to another constructor. The second constructor does not initialize the nonstatic data members by using member initializers. Polyspace flags the second constructor.

## Check Information

**Group:** Special Member Functions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-7-1

If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined

## Description

### Rule Definition

*If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined.*

### Rationale

Compilers implicitly define special member functions if these functions are declared as `=default` or left undefined. These implicitly defined functions are consistent, error-free, and do not require maintenance. If the implicitly defined special member functions are sufficient, then replacing them with user-defined functions makes the code error-prone and harder to maintain. Unless a class manages resources like a raw pointer or a POSIX file descriptor, the implicit definition of the special member functions might be sufficient. Avoid defining the special member functions when the implicit definitions are sufficient.

Default construction of `const` objects might cause a compilation failure if the nonstatic data members are not initialized during definition. The best practice is to initialize nonstatic data members in the class definition. Alternatively, initialize the `const` instance by using an empty initializer list. These practices enable the default constructor to correctly construct `const` instances of a class.

### Polyspace Implementation

Polyspace raises the checker if the user-defined special member functions of your class are the same as the implicitly defined special member functions. The implicit special member functions typically have these properties:

- The implicit default constructor have an empty body, an empty parameter list, and an empty initializer list.
- The implicit destructors have an empty body.
- The implicit copy/move constructor and assignment operators copy/move the base classes and nonstatic data members by using an initializer list. These functions does not perform any deep copy and does not move the data associated with a raw pointer.

For details about how implicitly defined special member function behave, see:

- Default constructor
- Destructor
- Copy constructor
- Copy Assignment operator
- Move constructor
- Move assignment operator

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Define Special Member Functions Only When Managing Resources

```
#include <cstdint>
#include <utility>
class A
{
public:
    A() : x(0), y(0) {}                                 // Compliant
    A(std::int32_t first, std::int32_t second)          // Compliant
    : x(first), y(second) {}
    A(const A& oth) : x(oth.x),y(oth.y){}               // Noncompliant
    A(A&& oth): x(std::move(oth.x)),y(std::move(oth.y)){} // Noncompliant
    ~A(){}                                              // Noncompliant


private:
    std::int32_t x;
    std::int32_t y;
};
class B
{
public:
    B() {}  // Noncompliant
    B(std::int32_t first, std::int32_t second)// Compliant
    : x(first), y(second)
    {}
    B(const B&) = default; // Compliant
    B(B&&) = default;     // Compliant
    ~B() = default;       // Compliant

private:
    std::int32_t x;
    std::int32_t y;
};
class D
{
public:
    D() : ptr(nullptr) {}  // Compliant - Managing a raw pointer
    D(B* p) : ptr(p) {}    // Compliant - Managing a raw pointer
    D(const D&) = default; // Requires user defined copy constructor
    D(D&&) = default;      // Requires user defined move constructor
    ~D() = default;        // Requires user defined destructor

private:
    B* ptr;
};
class E      // Compliant
{
};
```

In this example, the classes A, B, and D are defined.

- For the class A:
  - The user defined default constructor has a nonempty initializer list. Because this definition is different than an implicit default constructor, Polyspace does not report a violation.
  - Because this rule does not apply to nondefault constructors, Polyspace does not report a violation.
  - Because the user-defined copy and move constructors of A do not manage any resources and they can be defined as =default or left undefined, Polyspace reports violations on these functions.
- For the class B:
  - The default constructor of B is user-defined even though it does not manage any resources. Because this user-defined constructor can be defined as =default or left undefined, Polyspace reports a violation on it.
  - Because the other constructors of B are defined as =default, they are compliant..
- The class D contains a raw pointer to the class B. For the class D:
  - Because the user-defined constructor explicitly initializes the raw pointer ptr to nullptr, it is different than an implicit constructor. Polyspace does not report a violation.
  - Because the other special member functions are declared as =default, these functions do not violate this rule.

  When managing resources such as a raw pointer, the implicitly defined special member functions are typically incorrect. For instance, the implicit copy constructor of D performs a shallow copy of ptr, without duplicating the underlying resources. When managing raw pointers, the best practice is to define all the special member functions appropriately to avoid unexpected behavior.

### Initialize Nonstatic Data Members to Default Construct const Instances

```
class B_NC{
    public:
    //B_NC()=default;// Compile fail
    B_NC(){}         //Noncompliant
    int x;
};
class B_C{
    public:
    B_C() = default;//Compliant
    int x = 0;
};
const B_NC a;
const B_C b;
```

In this example, classes B_NC and B_C are defined and const instances of these classes are constructed. According to the C++ standard, a const object can be constructed by the implicit default constructor only if all the members of the object is const-default-constructible. The variable B_NC::x cannot be const-default-constructed. If you declare the constructor as =default, constructing a const instance of B_NC results in a compilation failure. To resolve the compilation failure, you might want to provide a user-defined constructor that is identical to the implicit constructor, such as the constructor B_NC::B_NC() in the preceding code. Though this code compiles, B_NC::B_NC() is noncompliant with this rule.

To resolve the compilation failure without violating this rule, initialize the nonstatic data members of the class in its definition, as shown in the definition of `B_C`. Because `B_C::x` can be const-default-constructed, a `const` instance of `B_C` can be constructed by the implicit default constructor. After initializing the nonstatic data members, set the constructor to `=default`.

## Check Information

**Group:** Special member functions
**Category:** Required, Automated

# Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-8-1

Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects

## Description

### Rule Definition

*Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects.*

### Rationale

The expected behavior of move and copy constructors is:

- They move or copy the base classes and data members.
- The move constructor sets the source object into a valid state.

Authoring move or copy constructors that have additional side effects might cause these issues:

- Performance: Move and copy constructors are frequently called by standard template library (STL) algorithms and containers. Performance overhead in these constructors caused by side effects can accumulate and affect the performance of your code.
- Unexpected behavior: Because compilers might omit calls to copy constructors to optimize the code, the number of times a copy constructor might be invoked is indeterminate. As a result, the side effects of a copy constructor might produce unexpected behavior.

### Polyspace Implementation

In the body of a copy or move constructor, Polyspace does not flag these operations:

- Copy or move assignments.
- Relational or comparison operations.
- Modification of the source object in a move operation.
- Calls to the function `std::swap` or equivalent user-defined `noexceot` swap functions. Polyspace identifies functions that these signatures as swap functions: `void T::swap(T&)` or `void [N::]swap(T&, T&)`. The first signature represents a member function of class `T` that takes one argument. The second signature represents a nonmember or static function in the namespace `N` that takes two arguments. The name `swap` can be case-insensitive and prefixed or postfixed by underscores.
- Assignment and modification of `static` variables.

Polyspace flags any other operations in a copy or move constructor as unwanted side effect. For instance, a call to a user-defined swap function is considered an unwanted side effect if the swap function is not `noexcept`. For a similar rule on copy and move assignment operator, see `AUTOSAR C++14 Rule A6-2-1`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Side Effects in Copy and Move Constructors

This code shows how Polyspace flags move and copy constructors that have side effects.

```
#include<cstdint>
#include <utility>
#include<iostream>
class B
{
public:
    // Implementation
    B(B&& oth) : ptr(std::move(oth.ptr)) // Noncompliant
    {
        oth.ptr = nullptr; // Does not have a side effect
        std::cout<<"Moved"; //Has a side effect
    }
    ~B()
    {
        delete ptr;
    }

private:
    std::int32_t* ptr;
};
class C
{
public:
    C(int t=0) : x(t) {}
    C(const C& oth): x(oth.x)  // Noncompliant
    {
        //...
        x = oth.x % 2;  // Has a side effect
        count++; //Not a side effect
    }

private:
    std::int32_t x;
    static std::int32_t count;
};
class D
{
public:

    D(const D& oth): x(oth.x)  // Noncompliant
    {
        D tmp(oth);
        _swap_(tmp);
    }
    void _swap_(D& rhs){ //Might raise exceptions
        //...
    }
private:
    std::int32_t x;
    static std::int32_t count;
};
```

- As a side effect, the move constructor of class B prints a string into the output stream. This side effect adds performance overhead to the move operation. Polyspace flags the move assignment operator and highlights the statement. Setting the moved-from object `oth.ptr` to `nullptr` is not a side effect.
- The copy constructor of the class C modifies the data member `x` of the source object. This side effect adds performance overhead. Unexpected change to data members during move and copy operations can make the code incompatible with the standard template library and introduce errors during development. Polyspace flags the copy assignment operator and highlights the statement `x = oth.x % 2`. Incrementing the static variable `count` is not a side effect.
- The copy constructor of the class `D` calls a user-defined swap function called `_swap_`. This swap function is not `noexcept`. If an exception is raised from `_swap_`, the exception is an unexpected side effect of the copy constructor. Polyspace flags the copy constructor as noncompliant with this rule. Use user-defined swap functions that are `noexcept`.

## Check Information

**Group:** Special member functions
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-8-2

User-defined copy and move assignment operators should use user-defined no-throw swap function

## Description

### Rule Definition

*User-defined copy and move assignment operators should use user-defined no-throw swap function.*

### Rationale

A naive copy or move assignment operator that is implemented without using a swap function might follow the pattern in this code:

```
class A{
    //...
    A & operator=(const A & rhs)
    {
        if (this != &rhs) // check for self assignment
        {
            // release resource in lhs

            // Allocate resource for modified lhs


            // Copy or move the resources from rhs to lhs

        }

        return *this;
    }
private:
    //resources
    int* mArray;
};
```

Such naive implementation of the copy or move assignment operator cannot provide strong exception safety because if any of the operations raises an exception, the left operand cannot be reverted back to its original state. The preceding pattern is also inefficient because it requires a check for self-assignment. Code duplication between such a copy or move assignment operator and a copy or move constructor makes the code difficult to maintain.

To resolve these issues, utilize user-defined `swap` functions that do not raise exceptions. Consider this pattern:

```
class A{
    //...
    A & operator=(A rhs)
    {
        Swap(*this,rhs);
    }
    friend void Swap(A& lhs, A& rhs) noexcept{
        //...
    }
private:
    //resources
    int* mArray;

};
```

This implementation of the copy or move assignment operator does not attempt allocation or deallocation of memory. Instead, It swaps the resources between the left and right operands by calling a user-defined `noexcept` function `Swap`. This `Swap` function might be implemented by utilizing the `std::swap` function. The benefits of this pattern are:

- Strong exception safety: This implementation of the copy or move assignment operator takes a temporary copy of the right operand by using the copy or move constructor and swaps the temporary copy with the left operand. Because the move and swap functions must be `noexcept`, only the copy operation might raise an exception. If this operator raises an exception, only the temporary copy of the right operand might be invalidated. The state of the right or the left operand remains untouched.
- Code reuse: In this implementation, the copy or move assignment operator reuses the copy or move constructor. The class-specific `swap` function can also be reused for implementing other algorithms.
- Efficiency: By eliminating the check against self-assignment, the operator is more efficient.

To implement a copy or move assignment operator, use user-defined `noexcet` swap functions.

### Polyspace Implementation

Polyspace flags a copy or move assignment operator if it does not contain at least one call to a user-defined swap function. Polyspace identifies functions that have these signatures as swap functions: `void T::swap(T&)` or `void [N::]swap(T&, T&)`. The first signature represents a member function of class `T` that takes one argument. The second signature represents a nonmember or static function in the namespace `N` that takes two arguments. The name `swap` can be case-insensitive and prefixed or postfixed by underscores.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Naive Implementation of Copy and Move Assignment Operators

```
#include <utility>
#include <string>
class B
{
  public:
    B& operator=(const B& oth) & { // Noncompliant
      if (this != &oth)
      {
        ptr1 = new std::int32_t(*oth.ptr1);
        ptr2 = new std::int32_t(
          *oth.ptr2); // Exception thrown here results in
        // a memory leak of ptr1
      }
      return *this;
    }
    B& operator=(B&& oth) & noexcept { // Noncompliant
      if (this != &oth)
      {
        ptr1 = std::move(oth.ptr1);
        ptr2 = std::move(oth.ptr2);
        oth.ptr1 = nullptr;
        oth.ptr2 = nullptr;
      }

      return *this;
    }
private:
    std::int32_t* ptr1;
    std::int32_t* ptr2;
};
```

In this example, the copy and move assignment operator for class B uses a naive implementation instead of a copy-and-swap implementation. The copy and move operator of B is inefficient and does not provide strong exception safety. Polyspace flags these operators as noncompliant.

#### Avoid Using Generic std::swap

```
#include <utility>
#include <string>
class C
{
  public:
    C(const C&) = default;
    C(C&&) = default;

    C& operator=(const C& oth) & {     //Noncompliant
      C tmp(oth);
      std::swap(ptr1, tmp.ptr1);
      return *this;
    }
    C& operator=(C&& oth) & {          // Noncompliant
      C tmp(std::move(oth));
      std::swap(ptr1, tmp.ptr1);
      return *this;
    }

  private:
    std::int32_t* ptr1;
};
```

In this example, the copy and move assignment operator for class C uses a copy-and-swap implementation, but uses the standard std::swap function instead of a class-specific, user-defined swap function. Because class C requires user-defined copy and move operator, it also require a user-defined swap function. Polyspace flags the operators as noncompliant.

#### Avoid swap Functions That Might Raise Exceptions

```
#include <utility>
#include <string>
```

```
class D
{
  public:
    D(const D&) = default;
    D(D&&) = default;
    D& operator=(const D& oth) & {     // Noncompliant
      D tmp(oth);
      _swap_(*this,tmp);
      return *this;
    }
    D& operator=(D&& oth) & {          // Noncompliant
      D tmp(std::move(oth));
      _swap_(*this,tmp);
      return *this;
    }
    //...
    friend void _swap_(D& lhs, D& rhs){ // swap function not noexcept
    //...
    }
};
```

In this example, the copy and move assignment operator for class `D` uses a `swap` function that is not `noexcept`. These operators do not provide strong exception safety. Polyspace flags them as noncompliant.

#### Avoid swap Functions With Unexpected Signature

```
#include <utility>
#include <string>

class E
{
  public:
    E(const E&) = default;
    E(E&&) = default;

    E& operator=(const E& oth) & {     // Noncompliant
      E tmp(oth);
      swap(*this,tmp);
      return *this;
    }
    E& operator=(E&& oth) & {          // Noncompliant
      E tmp(std::move(oth));
      swap(*this,tmp);
      return *this;
    }

    // Member function swap
    void swap(E& lhs, E& rhs) noexcept {
      std::swap(lhs.ptr1, rhs.ptr1);
      std::swap(lhs.ptr2, rhs.ptr2);
    }

  private:
    std::int32_t* ptr1;
    std::int32_t* ptr2;
};
```

In this example, the copy and move assignment operator for class `E` uses a `swap` function that takes two arguments. Because the swap function is defined as a nonstatic member function of `E`, Polyspace expects the `E::swap` function to have only one argument. Polyspace flags the copy and move operators of `E` because the swap function has an unexpected signature.

### Check Information

**Group:** Special member functions
**Category:** Advisory, Automated

## Version History

**Introduced in R2021a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-8-3

Moved-from object shall not be read-accessed

## Description

### Rule Definition

*Moved-from object shall not be read-accessed.*

### Rationale

Because the content of a source object is generally unspecified after a move operation, it is unsafe to perform operations that access the contents of the source object after a move operation. Accessing the contents of the source object after a move operation might result in a data integrity violation, an unexpected value, or an illegal dereferencing of a pointer.

Operations that make no assumptions about the state of an object do not violate this rule.

The C++ standard specifies that these move operations leave the source object in a well-specified state after the move:

- Move construction, move assignment, converting[22] move construction, and converting move assignment of `std::unique_ptr` type
- Move construction, move assignment, converting move construction, converting move assignment of `std::shared_ptr` type
- Move construction and move assignment from a `std::unique_ptr` of `std::shared_ptr` type
- Move construction, move assignment, converting move construction, and converting move assignment of `std::weak_ptr` type
- `std::move()` of `std::basic_ios` type
- Move constructor and move assignment of `std::basic_filebuf` type
- Move constructor and move assignment of `std::thread` type
- Move constructor and move assignment of `std: unique_lock` type
- Move constructor and move assignment of `std::shared_lock` type
- Move constructor and move assignment of `std::promise` type
- Move constructor and move assignment of `std::future` type
- Move construction, move assignment, converting move construction, and converting move assignment of `std::shared_future` type
- Move constructor and move assignment of `std::packaged_task` type

Because these move operations leave the source object in a well-specified state, accessing the source object after calling these functions is compliant with this rule.

[22] A converting constructor is a constructor that is not declared with the specifier `explicit`. See Converting constructor.

### Polyspace Implementation

Polyspace raises a flag if the source object is read after its contents are moved to a destination object by calling the `std::move` function explicitly. Polyspace does not flag accessing a source object if:

- The source object of an explicit move operation is of these types:
  - `std::unique_ptr`
  - `std::shared_ptr`
  - `std::weak_ptr`
  - `std::basic_ios`
  - `std::basic_filebuf`
  - `std::thread`
  - `std::unique_lock`
  - `std::shared_lock`
  - `std::promise`
  - `std::future`
  - `std::shared_future`
  - `std::packaged_task`
- The move operation is performed implicitly. For instance, the function `std::remove` might access the source object after an implicit move operation. Polyspace does not flag it. A best practice is to avoid such operations and use safer alternatives that prevent accidental access, such as `std::erase`.
- The source object is of a built-in base type, such as: `int`, `enum`, `float`, `double`, pointer, `std::intptr_t`, `std::nullptr_t`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Reading Source Object After Calling `std::move`

This example shows how Polyspace flags reading the source object after an explicit move operation.

```
#include<string>
#include<iostream>
void F1()
{
    std::string s1{"string"};
    std::string s2{std::move(s1)};
    // ...
    std::cout
    <<  // Noncompliant
    s1
    << "\n";
}
```

```
void F2()
{
    std::unique_ptr<std::int32_t> ptr1 = std::make_unique<std::int32_t>(0);
    std::unique_ptr<std::int32_t> ptr2{std::move(ptr1)};
    std::cout << ptr1.get() << std::endl; // Compliant by exception
}
void g(std::string v)
{
    std::cout << v << std::endl;
}

void F3()
{
    std::string s;
    for (unsigned i = 0; i < 10; ++i) {
        s.append(1, static_cast<char>('0' + i));  //Noncompliant
        g(std::move(s));
    }
}
void F4()
{
    for (unsigned i = 0; i < 10; ++i) {
        std::string s(1, static_cast<char>('0' + i)); // Compliant
        g(std::move(s));
    }
}
```

- In the function `F1`, string `s1` is explicitly moved to `s2` by calling `std::move`. After the move operation, the function attempts to read `s1`. Polyspace flags this attempt of reading a source object after an explicit move.
- In the function `F2`, the unique pointer `ptr1` is explicitly moved to `ptr2`. Because the `std::unique_ptr` remains in a specified state after the move, reading a source unique pointer after an explicit move is compliant with this rule.
- In the function `F3`, the string `s` is explicitly moved, and then it is read by the `std::string::append` function. Polyspace flags this attempt of reading a source object after an explicit move.
- In the function `F4`, the string `s` is explicitly moved. In each iteration of the loop, `s` is initiated to specific content before the move operation is triggered. As a result, the state of `s` is specified before the object is accessed. This method of accessing the source object after a move operation is compliant with this rule.

### Check Information

**Group:** Special member functions
**Category:** Required, Partially automated

## Version History

**Introduced in R2021a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `CERT C++: EXP63-CPP`

**Topics**
"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-8-4

Move constructor shall not initialize its class members and base classes using copy semantics

## Description

### Rule Definition

*Move constructor shall not initialize its class members and base classes using copy semantics.*

### Rationale

In C++, move operations transfer the ownership of resources rather than duplicating the resources themselves from a source object to a target object. Because move constructors do not duplicate resources, these constructors are faster than copy constructors. Consider this code where the object `CopyTarget` is copy-constructed and the object `MoveTarget` is move-constructed from the object `Source`.

```
class BigData{
    //...
    BigData(BigData&&){  //Move Constructor
        //...
    } copy constructed
    BigData(const BigData&){  //Copy Constructor
        //...
    }
private:
    std::map<int, std::string> BigBook;
};

int main(){
    BigData Source;
    BigData CopyTarget = Source;
    BigData Movetarget = std::move(Source);
    //...
}
```

When copy-constructing `CopyTarget`, the compiler duplicates the resource `Source::BigBook` from `Source` to `CopyTarget`. After the copy-construction, both of these objects have a copy of the resource `BigBook`. When move-constructing `Movetarget`, the compiler transfers the ownership of the resource `Source::BigBook` to `MoveTarget`. Because move-construction does not duplicate the resource physically, it is faster than copy-construction.

Move-construction is an optimization strategy. You expect that move-construction is cheaper and faster than copy-construction. Copy-initializing data members and base classes can make a move constructor slow and inefficient, which reduces program performance. Developers expect that move-construction uses move semantics only. Unexpectedly using copy semantics in move constructors might introduce resource leaks and inconsistency in future development. When authoring move constructors, initialize data members and base classes by using move semantics. You can copy-initialize scalar data members without violating this rule.

You might use `std::move()` to implement move semantics in your code. When you use `std::move()` to move objects, declare the objects or data members without the qualifier `const`. For more information, see `AUTOSAR C++14 Rule A18-9-3`.

### Polyspace Implementation

When a move constructor does not use move semantics to initialize nonscalar data members and base classes, Polyspace flags its declaration. For instance, if a move constructor initializes the base class by using the default constructor instead of the move constructor, Polyspace flags the declaration of the move constructor.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Move-Initialize Nonscalar Data Members and Base Classes in Move Constructors

This example shows how Polyspace flags move constructors that uses copy-initialization and default construction.

```
#include<cstdint>
#include<string>
#include<map>
#include <vector>
class BigData{
public:
    BigData()=default;
    //...
    BigData(BigData&& oth): //Compliant
    BigBook(std::move(oth.BigBook)),
    Length(oth.Length)
    {
        //...
    }

private:
    std::map<int, std::string> BigBook;
    int Length;
};

class slowBigData{
    //...
    slowBigData(slowBigData&& oth): //Noncompliant
    BigBook(oth.BigBook),
    Length(oth.Length)
    {
        //...
    }

private:
    std::map<int, std::string> BigBook;
    int Length;
};

class BigData2: public BigData{
    //...
    BigData2(BigData2&& oth):BigData() //Noncompliant
    {
        BigVector = std::move(oth.BigVector);
        //...
    }

private:
    std::map<int, std::string> BigBook;
    std::vector<int> BigVector;
    int Length;
};

class BigData3: public BigData{
    //...
    BigData3(BigData3&& oth):BigData(std::move(oth)) //Compliant

    {
        str = std::move(oth.str);
        //...
    }

private:
    std::map<int, std::string> BigBook;
    int Length;
    std::string str;
};
```

- The move constructor of the class `BigData` initializes the data member `BigBook` by using move semantics. The move constructor initializes the scalar member `Length` by using copy semantics. This move constructor is compliant because copying scalar data members does not violate this rule.
- The move constructor of the class `slowBigdata` initializes the data members by using copy semantics. This move constructor violates the rule and Polyspace flags the declaration of the move constructor.
- The move constructor of the class `BigData2` invokes the default constructor of the base class, which might make the code slow and inefficient. Polyspace flags the declaration of this move constructor. The move constructor of the class `BigData3` invokes the move constructor of the base class. This move constructor is compliant with this rule.

## Check Information

**Group:** Special member functions
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-8-5

A copy assignment and a move assignment operators shall handle self-assignment

## Description

### Rule Definition

*A copy assignment and a move assignment operators shall handle self-assignment.*

### Polyspace Implementation

Reports when copy assignment body does not begin with “`if (this != arg)`”

A violation is not raised if an empty `else` statement follows the `if`, or the body contains only a return statement.

A violation is raised when the `if` statement is followed by a statement other than the return statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Special Member Functions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A12-8-6

Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class

## Description

### Rule Definition

*Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class.*

### Rationale

Pointers to derived classes are type-compatible with pointers to base classes. A pointer can be an object of the base class while pointing to an object of the derived class. When such an object is copied, the base copy constructor is invoked and the copied object has only the base part of the original object. To avoid inadvertent slicing during copy and move, suppress these operations in the base class by:

- Declaring copy and move constructors and copy assignment and move assignment operators as `protected`.
- Defining copy and move constructors and copy assignment and move assignment operators as "`=delete`".

### Polyspace Implementation

Polyspace flags these special member functions of a base class when they are not declared `protected` or defined as `=delete`:

- Copy constructor
- Move constructor
- Copy assignment operator
- Move assignment operator

Polyspace indicates which special member function violates this rule.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Suppress Copy and Move Operations in Base Classes

```
#include <cstdint>
#include <memory>
#include <utility>
#include <vector>
class A
```

```
{
  public:
    int base_var;
    A() = default;
    A(A const&) = default;                //Noncompliant
    A(A&&) = default;                     //Noncompliant
    virtual ~A() = 0;
    A& operator=(A const&) = default;     //Noncompliant
    A& operator=(A&&) = default;          //Noncompliant
};
class B : public A
{
    int derived_var;
};
class C //
{
  public:
    int base_var;
    C() = default;
    virtual ~C() = 0;

  protected:
    C(C const&) = default;                //Compliant
    C(C&&) = default;                     //Compliant
    C& operator=(C const&) = default;     //Compliant
    C& operator=(C&&) = default;          //Compliant
};
class D : public C
{
    int derived_var;
};
class E
{
  public:
    int base_var;
    E() = default;
    virtual ~E() = default;
    E(E const&) = delete;                   //Compliant
    E(E&&) = delete;                        //Compliant
    E& operator=(E const&) = delete;        //Compliant
    E& operator=(E&&) = delete;             //Compliant
};

class F : public E
{
    int derived_var;
};
void Fn1() noexcept
{
  B obj1;
  B obj2;
  A* ptr1 = &obj1;
  A* ptr2 = &obj2;
  *ptr1 = *ptr2;             // Partial assignment only
  *ptr1 = std::move(*ptr2); // Partial move only
  D obj3;
  D obj4;
  C* ptr3 = &obj3;
  C* ptr4 = &obj4;
  // *ptr3 = *ptr4; // Compilation error
  // *ptr3 = std::move(*ptr4); // Compilation error
  F obj5;
  F obj6;
  E* ptr5 = &obj5;
  E* ptr6 = &obj6;
  // *ptr5 = *ptr6; // Compilation error
  // *ptr5 = std::move(*ptr6); // Compilation error
}
```

The Class A is a base class with default copy and move constructors and default copy and move assignment operators. The class B is derived from A and has a variable derived_var that is absent in A. In Fn1(), two pointers ptr1 and ptr2 are created. They are objects of the base class A, but point to obj1 and obj2 respectively, which are objects of the derived class B. The assignment A *ptr = &obj1; is an example of polymorphic behavior where you can declare a pointer of the base class and assign objects of any derived class to it.

Because ptr1 and ptr2 are objects of the base class A, the copy operation in *ptr1 = *ptr2 invokes the default copy assignment operator of class A. The default semantics copies only the base part of obj2 into obj1. That is, obj2.derived_var is not copied into obj1.derived_var. Similarly, the ownership of obj2.derived_var is not moved to obj1 by the move operation in *ptr1 = std::move(*ptr2). To avoid inadvertent slicing, suppress the copy and move operations in the base class of a class hierarchy. Polyspace flags the copy and move functions in base class A because these functions are neither declared as protected nor defined as =delete.

In class C, the copy and move functions are suppressed by declaring the copy and move constructors and copy assignment and move assignment operators protected. In class E, the copy and move operations are suppressed by declaring these special member functions as =delete. If you invoke the copy or move operations of these base classes, the compiler generated an error. The definitions of the base classes C and E are compliant with this rule.

### Check Information

**Group:** Special member functions
**Category:** Required, Automated

## Version History

**Introduced in R2020a**

### See Also

Check AUTOSAR C++ 14 (-autosar-cpp14)

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A12-8-7

Assignment operators should be declared with the ref-qualifier &

## Description

### Rule Definition

*Assignment operators should be declared with the ref-qualifier &.*

### Rationale

You can use ref-qualifiers to specify whether a function or operator applies to lvalues or rvalues. Functions or operators that apply to lvalues have the ref-qualifier &. Functions and operators that apply on rvalues have the ref-qualifier && at the end of their declaration.

Built-in assignment operators in C++ accept only lvalues as input parameters. If user-defined assignment operators take both rvalue and lvalue as input parameters, it can cause confusion and errors. Consider this code where the user-defined assignment operator for the class `obj` accepts both rvalues and lvalues as input parameters.

```
class obj{
    obj& operator=(Obj const&){
        //...
        return *this;
    }
    //...
};

int main(){
    int i,j,k;
    obj a,b,c;

    if((i+j)=k) // compilation error
    //...
    if((a+b)=c) // silent error
    //...
}
```

- In the first `if` statement, the equal-to operator (==) is written as an assignment operator (=) because of a typographical error. Because the built-in assignment operator for `int` does not accept rvalues as input, the statement `(i+j) = k` causes a compilation error.
- The condition for the second `if` statement contains a similar error. Because the user-defined assignment operator for class `obj` accepts both lvalues and rvalues as input, the statement `(a+b) = c` compiles without error. The `if` block executes unexpectedly, resulting in a silent bug.

To avoid errors and confusion, specify that assignment operators take only lvalues as input parameters by adding the ref-qualifier & to their declaration.

### Polyspace Implementation

Polyspace flags user-defined assignment, compound assignment, increment, and decrement operators when:

- They do not have the ref-qualifier & in their declaration.
- They are member functions of a class.
- They are not declared as `= delete`.

Because ref-qualifiers are applicable only to nonstatic member functions, this rule does not apply to nonmember assignment operators.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Declare Assignment Operators by Using Ref-Qualifier &

This example shows how Polyspace flags assignment operators and increment or decrement operators when their declarations do not specify the ref-qualifier &.

```
#include<cstdint>
class Obj
{
public:
    Obj() = default;
    Obj& operator=(Obj const&) & = default; //Compliant
    Obj& operator=(Obj&&) & = default;      //Compliant
    Obj& operator++() & noexcept;           //Compliant
    Obj& operator--()  noexcept;            //Noncompliant
    Obj& operator<<=(Obj const&) noexcept;  //Noncompliant
    Obj& operator>>=(Obj const&) & noexcept;//Compliant
    Obj& operator+=(Obj const&)&;           //Compliant
    Obj& operator-=(Obj const&);            //Noncompliant
    Obj& operator*=(Obj const&)= delete;    //Compliant
    Obj& operator+(Obj const&)&;            //Compliant
};

Obj& operator|=(Obj& f,const std::int32_t i) // Rule does not apply
{
    return f;
}

Obj& Obj::operator+=(Obj const&) &  // Polyspace flags the declaration
{
    return *this;
}
Obj F1() noexcept
{
    return Obj{};
}
int main()
{
    Obj c;
    //F1() += c; // Compilation Error
    //F1() = c; // Compilation Error
    F1() -= c; // Silent Bug
}
```

In `main()`, the assignment operators `+=`, `-=`, and `=` are used with an rvlaue input. Because the declarations of the operators `+=` and `=` specify the ref-qualifier `&`, using these operators with an rvalue input results in a compilation failure. The operator `-=` is declared without the reference qualifier `&`. Using this operator with an rvalue input creates a silent bug.

- Polyspace flags nondeleted member assignment operators, increment operators, and decrement operators that do not specify the ref-qualifier `&` in their declarations.
- When a member assignment operator is declared without the reference qualifier `&` in a class and defined elsewhere, Polyspace flags the declaration.
- Polyspace does not flag nonmember operators without the ref-qualifier `&` because this rule applies only to nonstatic member functions.
- Polyspace does not flag deleted operators because using the ref-qualifier `&` on a deleted operator has no impact on the code.

## Check Information

**Group:** Special member functions
**Category:** Advisory, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A13-1-2

User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters

## Description

### Rule Definition

*User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.*

### Rationale

Since C++11, you can add suffixes to literals that convert numeric values under the hood. For instance, in code where you perform all calculations in a common unit, you can leave unit conversions to dedicated operators and simply use literal suffixes for the units when defining constant values.

In this example, the literal suffixes `_m` and `_km` resolve to calls to `operator"" _m()` and `operator"" _km()` respectively. The operators ensure that all values are converted to the same unit.

```
constexpr long double operator"" _m(long double metres) {
    return metres;
}

constexpr long double operator"" _km(long double kilometres) {
    return 1000*kilometres;
}
...
long double minSteps = 100.0_m;
long double interCityDist = 100.0_km;
```

User defined literal suffixes must begin with an underscore (_). Literal suffixes not beginning with underscore are reserved for the Standard Library.

### Polyspace Implementation

The rule checker flags definitions of the form:

```
operator "" suffix
```

where *`suffix`* does not begin with an underscore or following the underscore, contains characters other than letters (numbers, special characters, and so on).

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Overloading

**Category:** Required, Automated

## Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A13-1-3

User defined literals operators shall only perform conversion of passed parameters

## Description

### Rule Definition

*User defined literals operators shall only perform conversion of passed parameters.*

### Rationale

User-defined literal operators are expected to simply convert their parameters to the operator return type. If your operators perform operations other than this conversion, reviewers or other developers who use the operator might find this behavior unexpected.

For instance, the operator `_km` is expected to convert a `long long int` parameter to the type `Distance`:

```
Distance operator"" _km(long long int param) {
  ...
}
```

Any other operation performed by this operator might be unexpected.

### Polyspace Implementation

The checker flags user-defined literal operators that have one of these issues:

- The operator has a return type `void`. Such an operator cannot return the converted parameter.
- The operator does not even use its parameter, let alone convert its parameter to the operator return type.
- The operator performs an operation that has or might have side-effects. Such an operator defies the expectation that a user-defined literal operator *only* converts its parameter. For instance, the operator might update a global or static variable (definite side-effect) or call another function (possible side-effect). In the latter case, look into the callee function body to make sure it does not have side effects.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### User-defined Literal Operator with Side Effects

```
#include <cstdint>

struct Distance {
   unsigned long long int distparam;
   constexpr explicit Distance(unsigned long long int v) : distparam(v) {}
};

constexpr Distance operator"" _m(unsigned long long int distparam) //Compliant
{
   return Distance(distparam);
}

static void updateNumberOfConversions() {
   static std::int32_t numberOfConversions = 0;
   numberOfConversions++;
}

constexpr Distance operator"" _km(unsigned long long int distparam) //Noncompliant
{
   updateNumberOfConversions();
   return Distance(distparam);
}
```

In this example, the operator `_m` simply converts its parameter to the return type `Distance` and does not violate the rule. The operator `_km`, in addition to converting its parameter, also calls the function `updateNumberOfConversions`, which modifies a static variable. Therefore, the operator `_km` has additional side effects and violates the rule.

## Check Information

**Group:** Overloading
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A13-2-1

An assignment operator shall return a reference to "this"

## Description

### Rule Definition

*An assignment operator shall return a reference to "this".*

### Polyspace Implementation

The following operators should return `*this` on method, and `*first_arg` on plain function:

- `operator=`
- `operator+=`
- `operator-=`
- `operator*=`
- `operator >>=`
- `operator <<=`
- `operator /=`
- `operator %=`
- `operator |=`
- `operator &=`
- `operator ^=`
- Prefix `operator++`
- Prefix `operator--`

Does not report when no return exists.

No special message if type does not match.

Messages in report file:

- An assignment operator shall return a reference to `*this`.
- An assignment operator shall return a reference to its first arg.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Overloading
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A13-2-2

A binary arithmetic operator and a bitwise operator shall return a "prvalue"

## Description

### Rule Definition

*A binary arithmetic operator and a bitwise operator shall return a "prvalue".*

### Rationale

Binary arithmetic operators such as `operator+()` and bitwise operators such as `operator|()` must return an object of type `T` by value without qualifiers (and not references, `T&`, or pointers ,`T*`). This requirement makes the return types consistent with the implementation of the same operators in the C++ Standard Library. A developer familiar with the standard operators can easily work with the overloaded forms of the operators.

A prvalue or pure rvalue is an object that does not have a name, cannot be pointed to or referenced, but can still be moved from. For instance, the result of a call to a function that returns by value is a prvalue.

### Polyspace Implementation

The checker flags implementations of binary and bitwise operators that return:

- A type with a qualifier such as `const` or `volatile`.
- A pointer or reference to another type.

Operators flagged by the checker include:

- Binary operators such as `operator+()`, `operator*()`, `operator/()`, and so on.
- Bitwise operators such as `operator&()`, `operator|()`, `operator<<()`, and so on.

  Note that the checker does not show violations on `operator<<()` and `operator>>()` that return `std::basic_istream`, `std::basic_ostream`, or `std::basic_iostream` types.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant and Compliant Binary Operators

```
#include <cstdint>

class A
{
};
class B
```

```
{
};

A* operator+(A const&, A const&) noexcept //Noncompliant
{
  return new A{};
}
B operator+(B const&, B const&) noexcept //Compliant
{
  return B{};
}

const A operator*(A const&, A const&) noexcept //Noncompliant
{
  return A{};
}
B operator*(B const&, B const&) noexcept //Compliant
{
  return B{};
}

std::int32_t* operator/(A const&, A const&) noexcept //Noncompliant
{
  return 0;
}
std::int32_t operator/(B const&, B const&) noexcept //Compliant
{
  return 0;
}
```

In this example, the operator overloads that take operands of type `A` return objects that are not prvalues. Hence, these operators violate the rule. For instance:

- `operator+` and `operator/` return pointers to objects.
- `operator*` returns a `const`-qualified object.

The equivalent operator overloads for type `B` return objects by value without qualifiers and comply with the rule.

## Check Information

**Group:** Overloading
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A13-2-3

A relational operator shall return a boolean value

## Description

### Rule Definition

*A relational operator shall return a boolean value.*

### Rationale

The return value from relational operators of the C++ Standard Library can be directly checked to see if a relation is true or false. Overloads of the relational operator must be consistent with this usage. Otherwise, users of the overloaded relational operator might see unexpected results. See example below.

### Polyspace Implementation

The checker flags overloads of relational operators that do not return a value of type `bool`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Relational Operator Not Returning Boolean Value

```
class aClass {
    int val;
public:
    aClass(int initVal) {
        val = initVal;
    }
    bool operator<=(aClass const& comparingObj )  noexcept{ //Compliant
        return(this->val <= comparingObj.val);
    }
    int operator>=(aClass const& comparingObj ) noexcept { //Noncompliant
        return(this->val <= comparingObj.val? -1:1);
    }
};

void func() {
    aClass anObj(0), anotherObj(1);
    if(anObj <= anotherObj) {
        /* Do something */
    }
    if(anObj >= anotherObj) {
        /* Do something else */
    }
}
```

In this example, the overload of `operator<=` returns a boolean value but the overload of `operator>=` does not return a boolean value. However, in function `func`, the operators <= and >= are used as if a boolean value is returned from the overloaded operators. Because the overload of `operator>=` does not return the value zero, the second `if` statement is always true, a result that you might not expect.

## Check Information

**Group:** Overloading
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A13-3-1

A function that contains "forwarding reference" as its argument shall not be overloaded

## Description

### Rule Definition

*A function that contains "forwarding reference" as its argument shall not be overloaded.*

### Rationale

Suppose that you define a template function `func` with a forwarding reference parameter `T&&` like this:

```
template <typename T> void func(T&& param) {}
```

Suppose that you overload this template function with another function:

```
void func(int param) {}
```

When the function `func` is called, it is difficult to tell whether the call resolves to the template function or the overload, without working through the intricacies of the overload resolution mechanism. A developer or reviewer can easily mistake which function is called after the overload resolution. For instance, the function call:

```
short var;
//...
func(var);
```

resolves to the template function because it is an exact match after template instantiation, but a developer or reviewer might think that the overload is called because the type `short` promotes to `int`.

To avoid this issue, do not overload on template functions that take forwarding references. For template constructors, you can constrain the constructors to not match the overloads (using `std::enable_if`). When constrained this way, there is no scope for confusion between the template constructor and its overloads.

### Polyspace Implementation

The checker flags definitions of template functions that contain forwarding references (template parameters with type `T&&`) if those functions are also overloaded. Events below the checker result show the locations of the overloads. If an overload is an implicitly defined member function such as a constructor, the corresponding event points to the containing class.

The checker shows you all template functions with forwarding references that are overloaded. If you determine that an overload cannot cause confusion, add a comment to your result or code to avoid another review. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” or “Address Results in Polyspace Access Through Bug Fixes or Justifications”.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Template Function with Forwarding Reference Overloaded

```
#include <cstdint>

template <typename T>
void func(T&& t) noexcept(false) //Noncompliant
{
}

void func(
  std::int32_t&& t) noexcept
{
}
```

In this example, the function `func(std::int32_t &&)` overloads the template function with the same name, and violates the rule.

### Template Constructor Overloaded by Implicit Constructor

```
#include <type_traits>

class A
{
public:
    template<typename T>
    A(T &&value) {} //Noncompliant
};

class B
{
public:
    template<typename T,
    std::enable_if_t<! std::is_same<std::remove_cv_t<
    std:: remove_reference_t<T>>, B>::value> * = nullptr>
    B(T &&value) {} //Compliant
};

A getObjA();
B getObjB();

void func() {
    A objA = getObjA();
    B objB = getObjB();
}
```

In this example, class `A` has a template constructor that is overloaded by the implicit move constructor of class `A`. The overloading violates the rule.

Class `B` circumvents this problem by constraining the template constructor to not match the implicit constructor. In this example, the implicit constructor move constructor is called when the function `getObjB` returns an object of type `B`.

## Check Information

**Group:** Overloading
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A13-5-1

If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented

## Description

### Rule Definition

*If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.*

### Rationale

Typically, you overload the subscript operator `operator[]` to provide read and write access to individual elements of an array or similar structure contained in a class. If you implement a non-`const` overload of `operator[]`, you must also implement a `const` version of this overload. Otherwise, you cannot use `operator[]` to read elements of a `const` object.

This rule allows the implementation of a `const` overload of `operator[]` for read-only access without the corresponding non-`const` overload.

### Polyspace Implementation

Polyspace flags the definition of the non-`const` member function if no corresponding `const` version of the member function is implemented.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### `const` Version of non-`const` Member Function not Implemented

```
#include <memory>
#include <iostream>

class MyList
{
private:
    static constexpr std::int32_t maxSize = 10;
    std::int32_t container[maxSize];

public:
    std::int32_t& operator[](std::int32_t index) //compliant, non-const version
    {
        return container[index];
    }
    const std::int32_t& operator[](std::int32_t index) const //compliant, const version
    {
        return container[index];
    }
};
```

```
class MyList_nc
{
private:
    static constexpr std::int32_t maxSize = 10;
    std::int32_t container[maxSize];

public:
    std::int32_t& operator[](std::int32_t index) //non-compliant, non-const version only
    {
        return container[index];
    }

};

void func() noexcept
{
    MyList list;
    list[2] = 3; // Uses non-const version of operator[]
    std::cout << list[2] << std::endl;

    const MyList clist = {};
    std::cout << clist[2] << std::endl; // Uses const version of operator[]


}
```

In this example, the overloads of `operator[]` in class `MyList` are compliant because both the `const` and non-`const` versions of the overload are implemented. In class `MyList_nc`, the member function is not compliant because only the non-`const` version was implemented.

## Check Information

**Group:** Overloading
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A13-5-2

All user-defined conversion operators shall be defined explicit

## Description

### Rule Definition

*All user-defined conversion operators shall be defined explicit.*

### Rationale

If you do not define a user-defined conversion operator with the `explicit` specifier, compilers can perform implicit and often unintended type conversions from the class type with possibly unexpected results.

The implicit conversion can occur, for instance, when a function accepts a parameter of a type different from the class type that you pass as argument. For instance, the call to `func` here causes an implicit conversion from type `myClass` to `int`:

```
class myClass {} {
  ...
  operator int() {...}
};
myClass myClassObject;

void func(int) {...}
func(myClassObject);
```

### Polyspace Implementation

The checker flags declarations or in-class definitions of user-defined conversion operators that do not use the `explicit` specifier.

For instance, `operator int() {}` can convert variable of the current class type to an `int` variable both implicitly and explicitly but `explicit operator int() {}` can only perform explicit conversions.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Missing `explicit` Keyword on Conversion Operator

```
#include <cstdint>

class MyClass {
public:
    explicit MyClass(int32_t arg): val(arg) {};
    operator int32_t() const { return val; }  //Noncompliant
    explicit operator bool() const {   //Compliant
        if (val>0) {
          return true;
        }
        return false;
     }
private:
    int32_t val;
};

void useIntVal(int32_t);
void useBoolVal(bool);

void func() {
    MyClass MyClassObject{0};
    useIntVal(MyClassObject);
    useBoolVal(static_cast<bool>(MyClassObject));
}
```

In this example, the conversion operator `operator int32_t()` is not defined with the `explicit` specifier and violates the rule. The conversion operator `operator bool()` is defined explicit and does not violate the rule.

When converting to a `bool` variable, for instance, in the call to `useBoolVal`, the `explicit` keyword in the conversion operator ensures that you have to perform an explicit conversion from the type `MyClass` to `bool`. There is no such requirement when converting to an `int32_t` variable. In the call to `useIntVal`, an implicit conversion is performed.

## Check Information

**Group:** Overloading
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A13-5-3

User-defined conversion operators should not be used

## Description

### Rule Definition

*User-defined conversion operators should not be used.*

### Rationale

User-defined conversion operators might be called when you neither want nor expect them to be called, which can result in unexpected type conversation errors. For instance, in this code snippet, the user-defined conversion operator converts type `customType` to `double` to allow mixed mode expressions:

```
class customType
{
    public:
    customType(int base, int exponent);
    //....
    operator double() const; // Conversion operator, convert customType to double
};

customType var1(2,5);
double var2 = 0.5 * var1; //Conversion operator called, converts var1 to double
```

While this conversion might be expected, if you attempt to print `var1` by using `cout << var1;` without defining `operator <<` for `customType` objects, the compiler uses the conversion operator to implicitly convert and print `var1` as a `double`.

To avoid these unexpected conversions, replace the conversion operator with an equivalent function. The function must then be called explicitly. If you cannot avoid using conversion operators in your application, see rule `AUTOSAR C++14 Rule A13-5-2`.

### Polyspace Implementation

Polyspace flags all calls to conversion operators.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of a User-Defined Conversion Operator

```
class customType
{
public:
    customType(int base, int exponent): b(base), exp(exponent) { /* ...*/}
    //....
    operator double() const;
    double as_double() const {/* ...*/}

private:
    int b; //base
    int exp; //exponent
};

int func(void)
{

    customType var1(2, 5);
    double var2 = 0.5 * var1; //Non-compliant
    double var3 = 0.5 * var1.as_double(); // Compliant

    return 0;
}
```

In this example, the conversion of `var1` to a `double` in the declaration of `var2` uses conversion operator `customType::operator double`. This conversion is non-compliant because it uses a user-defined conversion operator.

The type conversion in the declaration of `var3` is compliant because it uses a function to handle the conversion, and this function must be called explicitly. This ensures that the conversion is expected.

## Check Information

**Group:** Overloading
**Category:** Advisory, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `AUTOSAR C++14 Rule A13-5-2`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A13-5-4

If two opposite operators are defined, one shall be defined in terms of the other

## Description

### Rule Definition

*If two opposite operators are defined, one shall be defined in terms of the other.*

### Rationale

Opposite operators have the same argument and return types but exactly complement each other. For instance, `operator==` checks if its arguments are equal and `operator!=` checks for the complementary relation, that is, inequality.

Defining opposite operators in terms of each other makes your code easier to maintain:

- If you define opposite operators independently of each other, each time you update one operator, you have to remember to update the other. For instance, if you update the definition of `operator==`. you have to also explicitly update the definition of `operator!=`.
- If you define opposite operators in terms of each other, updating only one operator is sufficient since that change implicitly updates the other operator.

### Polyspace Implementation

The checker raises a violation if one of the operators in the following pairs is not defined in terms of the other:

- `operator==`, `operator!=`
- `operator<`, `operator>=`
- `operator>`, `operator<=`

The checker flags one of the operators in the pair. The event list below the checker result shows the location of the other operator.

The checker considers two opposite operators as part of a pair only if they have the same argument and return types.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Opposite Operators Defined in Compliant and Noncompliant Ways

```
#include <cstdint>

class Info {
    public:
       explicit Info(std::uint32_t x, std::uint32_t y): x(x), y(y) {}
       friend bool operator==(const Info & lhs, const Info & rhs) {
           return ((lhs.x == rhs.x) && (lhs.y == rhs.y));
       }
       friend bool operator!=(const Info & lhs, const Info & rhs) { //Noncompliant
           return ((lhs.x != rhs.x) || (lhs.y != rhs.y));
       }
    private:
       std::uint32_t x;
       std::uint32_t y;
};

class Data {
    public:
       explicit Data(std::uint32_t x, std::uint32_t y): x(x), y(y) {}
       friend bool operator==(const Data & lhs, const Data & rhs) {
           return ((lhs.x == rhs.x) && (lhs.y == rhs.y));
       }
       friend bool operator!=(const Data & lhs, const Data & rhs) { //Compliant
           return !(lhs == rhs);
       }
    private:
       std::uint32_t x;
       std::uint32_t y;
};
```

In this example, the class `Info` defines the two opposite operators `operator==` and `operator!=` independently of each other. If you add another data member to the class or change to a different definition of equality, you have to explicitly update the definitions of both operators.

The class `Data` defines the same two operators, but `operator!=` is defined in terms of `operator==`. If you add a data member to the class later or change to a different definition of equality, you can change only the `operator==` operator. The `operator!=` operator will automatically reflect the change.

## Check Information

**Group:** Overloading
**Category:** Required, Automated

# Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A13-5-5

Comparison operators shall be non-member functions with identical parameter types and noexcept

## Description

### Rule Definition

*Comparison operators shall be non-member functions with identical parameter types and noexcept.*

### Rationale

Comparison operators must not compare objects that are of different types. If you pass objects of different types as arguments to a comparison operator, the operator must be able to convert one argument to the data type of the other.

Member functions have the inherent limitation that the implicit object parameter (the one referred to by the `this` pointer) cannot be converted to another data type. To support data type conversions when required, define comparison operators as non-member functions.

Comparison expressions are fundamental operations and must be noexcept. The comparison operators covered by this rule are:

- `==`
- `!=`
- `<`
- `<=`
- `>`
- `>=`

**Note** Declare comparison operators as `friend` to enable them to access internal data similar to a member function. This practice is allowed by the exception in rule A11-3-1.

### Polyspace Implementation

The checker flags comparison operators that are defined as member functions. The checker also flags non-member comparison operators that:

- Compare nonidentical parameter types, such as a `class` type and `int`.
- Are not declared with the `noexcept` specifier.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant Member Function Declaration

The declaration of `nonComp::operator::==` is noncompliant because the comparison operator is declared as a member function.

```
#include <cstdint>

class nonComp
{
  public:
      explicit nonComp(std::uint32_t d): m_d(d)
      {}
      bool operator ==(const nonComp& rhs) noexcept  //Non-compliant; member function
      {
          return m_d == rhs.m_d;
      }

  private:
      std::uint32_t m_d;
};


class Compliant
{
  public:
      explicit Compliant(std::uint32_t d): m_d(d)
      {}
      friend bool operator ==(Compliant const& lhs, Compliant const& rhs) noexcept
      {
          return lhs.m_d == rhs.m_d;
      }
  private:
      std::uint32_t m_d;
};
// Compliant; non-member, identical parameter types, noexcept
```

The class `Compliant` declares `operator::==` as a `friend`, so this comparison operator is compliant.

### Noncompliant Declaration with Different Types

The first declaration of `operator::==` compares two different data types, because of which this comparison operator is noncompliant.

```
#include <cstdint>

class nonComp
{
    using Self = nonComp;
};

class MemberFunc
{
    using Self = MemberFunc;
};

bool operator ==(const nonComp& lhs, //noncompliant; comparison operator for different data types.
                 const MemberFunc& rhs) noexcept
{
    return true;
}


bool operator ==(const nonComp& lhs, const nonComp& rhs) noexcept
{
    return true;
} //compliant; because it compares the same data types.
```

The second declaration of `operator::==` compares the same data types, so this comparison operator is compliant.

### Noncompliant Declaration That Is Not Noexcept

`nonComp::operator::==` is not declared as a noexcept, so this comparison operator is noncompliant.

```
#include<cstdint>
class nonComp
{
  public:
    explicit nonComp(std::uint32_t d): m_d(d)
    {}
    friend bool operator ==(nonComp const& lhs, //Noncompliant; member function isn't noexcept
                            nonComp const& rhs)
    {
        return lhs.m_d == rhs.m_d;
    }

  private:
    std::uint32_t m_d;
};


class Compliant
{
  public:
    explicit Compliant(std::uint32_t d): m_d(d)
    {}
    friend bool operator ==(Compliant const& lhs, Compliant const& rhs) noexcept
    {
        return lhs.m_d == rhs.m_d;
    }

  private:
    std::uint32_t m_d;
};
// Compliant; non-member, identical parameter types, noexcept
```

`Compliant::operator::==` is declared as a noexcept, so this comparison operator is compliant.

## Check Information

**Group:** Overloading
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A13-6-1

Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits

## Description

### Rule Definition

*Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits.*

### Rationale

Since C++14, you can introduce a separator `'` to separate digits in a digit sequence for better readability. For consistency across your code, follow this convention when entering the digit sequence separator:

- In decimal values, starting from the right, place the separator after every three digits, for instance, `3'000'000`.
- In hexadecimal values, starting from the right, place the separator after every two digits, for instance, `0xF'FF'0F`.
- In binary values, starting from the right, place the separator after every four digits, for instance, `0b1001'0011'0100`.

If you are consistent across your code, a developer or code reviewer can follow your code more easily and possibly estimate the order of magnitude of a value from the digit sequence separators.

### Polyspace Implementation

This checker follows the specifications of the AUTOSAR C++14 rule.

For integers, the checker starts checking from the right. For instance, the checker raises a violation on the value `45'30'00`, because starting from the right, the digit sequence separator appears after two digits instead of the expected three.

For floating-point numbers, the checker begins the check from the decimal point and proceeds outwards. The checker checks:

- The part before the decimal starting from the right.
- The part after the decimal starting from the left.

For floating-point numbers with a mantissa and exponent, the same rule as integers applies to the exponent. For instance, in the decimal notation, the checker checks exponents starting from the right and raises a violation if the digit sequence separators are placed, for instance, after every two digits instead of three.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Placement of Digit Sequence Separators in Integers

```
#include <cstdint>

std::uint32_t largeNum = 3'0000'0000; //Noncompliant
std::uint32_t smallerNum = 3'000'000;  //Compliant
std::uint32_t evenSmallerNum = 3'00'00; //Noncompliant

std::uint32_t largeHexNum = 0xFF'FF'FF'FF; //Compliant
std::uint32_t smallerHexNum = 0xFFF'FFF; //Noncompliant
```

In this example, the placement of digit sequence separators is compliant if the separators follow the expected convention:

- In decimal numbers, starting from the right, the separator is placed after every three digits.
- In hexadecimal numbers, starting from the right, the separator is placed after every two digits.

### Placement of Digit Sequence Separators in Floating-Point Numbers

```
#include <cstdint>

float PI = 3.1'415'926'53; //Noncompliant
float pi = 3.141'592'653; //Compliant;

float one_LB_to_KG = 0.45'359'237; //Noncompliant
float one_lb_to_kg = 0.453'592'37; //Compliant
```

In this example, the same floating-point number is assigned to two different variables but the placement of the digit sequence separators is different. The placement is compliant if the separators follow the expected convention:

- For digits after the decimal, starting from the left, the separator is placed after every three digits.
- For digits before the decimal, starting from the right, the separator is placed after every three digits.

  For floating-point numbers, the need for a digit sequence separator before a decimal is typically a rare occurrence. For instance, if you store a floating point-number in normalized form, the mantissa has only one digit before the decimal.

## Check Information

**Group:** Overloading
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A14-1-1

A template should check if a specific template argument is suitable for this template

## Description

### Rule Definition

*A template should check if a specific template argument is suitable for this template.*

### Rationale

A template defines the operations of a class or function for generic template types. If these operations require that the template types have specific characteristics, for instance the data type must be copy constructible, check the template arguments to ensure that they are suitable and have the required characteristics. Typically, you use `static_assert` assertions to perform this check at compile time, for instance, `static_assert(std::is_copy_constructible<T>)`.

### Polyspace Implementation

Polyspace flags template classes, structs, unions, and functions unless one of the following is true:

- The template contains at least one `static_assert` assertion, even if that assertion does not test the characteristics of the template parameters.
- The template is explicitly fully specialized even when it does not contain any `static_assert` assertions.

Polyspace does not flag template declarations.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Template Does Not Check Template Arguments

In this example, templates `specializedClass`, `func_def`, and `myStruct` are noncompliant because these templates do not do not contains any `static_assert` statement and they are not fully specialized. Polyspace reports violations of this rule on the noncompliant templates.

Polyspace does not flag:

- The class template `myTemplate` because it uses `static_assert` to check whether the template type is copy-constructible. If the instantiation of `myTemplate<myClass>` in function `F()` is uncommented, the assertion results in a compile-time error.
- The `struct` template `floatStruct` because this template uses a `static_assert` statement that checks if the input type is `float`.
- The class template `specializedClass<std::int32_t>` because it is explicitly fully specialized.

- The function template declaration `func_decl` because the template is not implemented.

```
#include <cstdint>
#include <type_traits>

template<typename T>
class myTemplate  // Compliant. Use of static_assert
{

    static_assert(std::is_copy_constructible<T>(),
                  "Template type not copy constructible.");
    //...

};
template<typename T>
class specializedClass //Non-compliant
{
};

template<typename T>
struct myStruct //Non-compliant
{
    //...
};
template<typename T>
struct floatStruct //Compliant
{
    static_assert(std::is_floating_point<T>::value, "Expecting a floating point type");
};
template<>
class specializedClass<std::int32_t>  // Compliant. Explicit full specialization
{
};

class myClass  //Not a template
{
public:
    myClass() = default;
    myClass(myClass const&) = delete;

};

class myOtherClass  //Not a template
{

};

template<typename T>
void func_decl(T const& obj) noexcept(false); // Compliant. Template declaration


template <typename T>
void func_def(T const& obj) noexcept(false)  //Non-compliant
{


}



void F()
{

    //myTemplate<myClass> a; // myClass is not copy constructible. Compile-time error.
    myTemplate<myOtherClass> b; //myOtherClass is copy constructible.
}
```

## Check Information

**Group:** Templates
**Category:** Advisory, Non-automated

# Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A14-5-2

Class members that are not dependent on template class parameters should be defined in a separate base class

## Description

### Rule Definition

*Class members that are not dependent on template class parameters should be defined in a separate base class.*

### Rationale

To access a member of a template class, you have to instantiate the template. If the member is not dependent on the template parameter, this instantiation step is not necessary. For instance, the members `anotherMember` and `someotherMember` of this template class `aClass` do not depend on the parameter `T`:

```
template <typename T>
class aClass {
   T aMember
   int anotherMember;
   int someotherMember
}
```

However, to access these members, you have to instantiate the template class `aClass`. To avoid the unnecessary template instantiation, do not include these members in the template declaration.

Including this member in the template declaration also causes unnecessary code bloat. Compilers generate a separate copy of a template class for each instantiation of a template. If a class member is not dependent on the template parameter, an identical copy of this member is created for each template instantiation.

### Polyspace Implementation

The checker flags data members of template classes that are not dependent on template parameters. The checker does not flag member functions.

If multiple data members of a template are flagged by this checker, create a base class for the template that aggregates these data members.

In some cases, you might not want to strictly adhere to this rule. For instance, if only a single member of the template class is not dependent on the template parameter, you might not want to create a separate base class for this member. If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Template Classes That Have Unnecessary Members

```
#include <cstdint>

template <typename T>
class aDataArray {
    T data[100];
    int32_t metadata[2]; //Noncompliant
    int32_t info;        //Noncompliant
};

class metadataArray {
    int32_t metadata[2];
    int32_t info;
};

template <typename T>
class anotherDataArray: public metadataArray { //Compliant
    T data[100];
};
```

In this example, the template class `aDataArray` includes data members `metadata` and `info` that are not dependent on the type parameter of the template, `T`. The template class `anotherDataArray` avoids the unnecessary instantiation. This class is derived from a base class `metadataArray`, which aggregates data members that are not dependent on a type.

## Check Information

**Group:** Templates
**Category:** Advisory, Partially automated

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A14-5-3

A non-member generic operator shall only be declared in a namespace that does not contain class (struct) type, enum type or union type declarations

## Description

### Rule Definition

*A non-member generic operator shall only be declared in a namespace that does not contain class (struct) type, enum type or union type declarations.*

### Rationale

This rule forbids placing generic operators in the same namespace as class (struct) type, enum type, or union type declarations. If the class, enum or union types are used as template parameters, the presence of generic operators in the same namespace can cause unexpected call resolutions.

Consider the namespace NS that combines a class B and a generic form of operator==:

```
namespace NS {
   class B {};
   template <typename T> bool operator==(T, std::int32_t);
}
```

If you use class B as a template parameter for another generic class, such as this template class A:

```
template <typename T> class A {
    public:
      bool operator==(std::int64_t);
}

template class A<NS::B>;
```

the entire namespace NS is used for overload resolution when operators of class A are called. For instance, if you call operator== with an int32_t argument, the generic operator== in the namespace NS with an int32_t parameter is used instead of the operator== in the original template class A with an int64_t parameter. You or another developer or code reviewer might expect the operator call to resolve to the operator== in the original template class A.

### Polyspace Implementation

For each generic operator, the rule checker determines if the containing namespace also contains declarations of class types, enum types, or union types. If such a declaration is found, the checker flags a rule violation on the operator itself.

The checker also flags generic operators defined in the global namespace if the global namespace also has class, enum or union declarations.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Generic Operator in Same Namespace as Class Type

```
#include <cstdint>

template <typename T> class Pair {
    std::int32_t item1;
    std::int32_t item2;
    public:
      bool operator==(std::int64_t ItemToCompare);
      bool areItemsEqual(std::int32_t itemValue) {
          return (*this == itemValue);
      }
};

namespace Operations {
    class Data {};
    template <typename T> bool operator==(T, std::int32_t); //Noncompliant
}

namespace Checks {
    bool checkConsistency();
    template <typename T> bool operator==(T, std::int32_t); //Compliant
}

template class Pair<Operations::Data>;
```

In this example, the namespace `Operations` violates the rule because it contains the class type `Data` alongside the generic `operator==`. The namespace `Checks` does not violate the rule because the only other declaration in the namespace, besides the generic `operator==`, is a function declaration.

In the method `areItemsEqual` in `template class Pair<Operations::Data>`, the == operation invokes the generic `operator==` method in the `Operations` namespace. The invocation resolves to this `operator==` method based on the argument data type (`std_int32_t`). This method is a better match compared to the `operator==` method in the original template class `Pair`.

## Check Information

**Group:** Templates
**Category:** Advisory, Automated

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A14-7-1

A type used as a template argument shall provide all members that are used by the template

## Description

### Rule Definition

*A type used as a template argument shall provide all members that are used by the template.*

### Rationale

A template can define operations on a generic type through one or more member variables or member functions. If the type that you use to instantiate the template does not provide all of the members that the template uses, your program might be ill-formed and might contain syntax or semantic errors.

For example, in the following code, template `TmplClass` declares a member function `someProperty()` but type `myType` does not. The instantiation of `TmplClass` by using `myType` is noncompliant and, as a result of the missing `someProperty()` function, `inst.func();` causes a compilation error.

```
template <typename T>
class TmplClass
{
public:
    void func()
    {
        T t;
        t.someProperty();
    }
};
struct myType {
};

void Instance() noexcept
{
    TmplClass<myType> inst; //Non-compliant
// inst.func(); //compilation error, struct myType has no member function someProperty()
}
```

### Polyspace Implementation

- Polyspace flags `class`, `struct`, or `union` template instantiations when the template parameter does not contain all of the members that the template uses.

  If you review results in the Polyspace desktop or web interfaces, in the template definition, the software highlights the members that are missing from the template parameter.
- Polyspace does not flag:
  - Function template instantiations.
  - Template instantiations that use an incomplete type as the template parameter.
  - Template instantiations that use a template parameter, where the missing member is a member type (nested type) or a member template.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Template Argument Does Not Provide All Members Used by Template

```
class IncompleteType;
class MyClass
{

    struct MyStruct {
        int Field1;
        int Field2;
    };

    struct MyStructArray {
        int Field3[12];
    };

public:
    MyStruct property1;
    int property2[10];
    MyStruct property3[10];
    MyStructArray property4;


};

template <typename T>
class TemplClass
{
public:
    void fooA()
    {
        T t;

        t.property1 = {1, 2};
        t.property2[5] = 5;
        t.property3[2].Field1 = 6;
        t.property4.Field3[4] = 10;

    }
};

class MyType
{
public:
    int rank;
};

class MyOtherType
{
public:
    int property;
};

template <typename T1, typename T2>
class TemplClass2Param
{
    void func()
    {
        T1 t1;
        T2 t2;
        t1.rank = 5;
        t2.rank2 = 6;
    }
};

void instantiate(void)
{
    TemplClass<MyClass> var; // Compliant
    TemplClass<IncompleteType> varFromIncomplete; // Compliant
    TemplClass2Param<MyType, MyOtherType> otherVar; //Non-compliant

}
```

In this example, the first instantiation of template `TmplCass` by using template parameter `MyClass` is compliant because `MyClass` provides all the members that the template uses.

Polyspace does not flag the second instantiation of `TmplCass` that uses incomplete type `IncompleteType`.

The instantiation of `TmplClass2Param` is noncompliant because one of the template parameters (`MyOtherType`) does not provide a member variable `rank2` that the template uses.

### Member Type or Member Template Missing from Template Parameter Used for Instantiation

```
class MyClass
{
  public:
    int var;
    void foo(int);
    typedef int nestedtype; // member type
    template<typename T>
    void tmpl_func() { }  //member template
};

template <typename T>
class TmplClass {
  public:
    void func() {
        T t;
        int j = t.var;
        t.foo(j);
        typename T::otherNestedType newvar; //member type not in MyClass
        newvar = 5;

        t.template other_tmpl_func<int>(); //member template not in MyClass
    }
};

void bar (void) {
    TmplClass<MyClass> instance;    // Compliant
}
```

In this example, template parameter `MyClass` does not have a member type `otherNestedType` or a member template `other_tmpl_function`, but the instantiation `TmplClass<MyClass> instance` is not flagged.

## Check Information

**Group:** Templates
**Category:** Required, Automated

# Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A14-7-2

Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared

## Description

### Rule Definition

*Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared.*

### Rationale

Observing this rule avoids situations where the behavior is undefined. For instance, if a compiler sees a partial specialization of a template *after* it has instantiated the template, the behavior is undefined. If you specialize a template in the same file as the template, this situation is less likely to occur.

You can also easily extend compile-time interfaces through specialization since the template and its specialization are in the same file and part of the same translation unit. The same reasoning applies to the requirement that a template specialization must be in the same file as the type for which the template is specialized.

### Polyspace Implementation

The checker checks each template specialization and raises a violation if:

- The specialization is not in the same file as the template that is specialized.
- The specialization is not in the same file as the user-defined type for which the template is specialized.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Templates
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A14-8-2

Explicit specializations of function templates shall not be used

## Description

### Rule Definition

*Explicit specializations of function templates shall not be used.*

### Rationale

Explicit specialization of function templates can cause unexpected issues with overload resolution in C++. Overload resolution:

- First searches for a non-template, plain-old-function that has a matching parameter list.
- If such a function is not available, overload resolution selects the closest matching function template.
- After a function template is selected, the compiler searches for a suitable specialization of the template.

Specializing a template does not change the order of the overload resolution process, which can result in confusing and unexpected behavior. Consider code snippet:

```
//(a) base template
template<class T> void f( T );

//(b) specialization of (a)
template<> void f<>(int*);
//...

//(c) overloads (a)
template<class T> void f( T* );

//...
main(){
    int *p;
    f( p );
}
```

When `f()` is called with an `int*` in `main()`, you might expect the specialization for `int*`, marked (b), to be called. The compiler resolves the call to `f()` as follows:

1. The compiler searches for a plain-old-function with input type `int*`.
2. Because there is no such function, the compiler searches for a function template that has the closest matching parameter list.
3. The template (c), which takes a generic pointer as input, is the closest match for `f(int*)`.
4. The template (c) is not specialized. The overload resolution process stops and calls the template in (c).

Even though a specialized template for `int*` type input is defined in (b), the overload resolves to the template in (c) instead, which can be unexpected.

When you specialize an overloaded function template, the overload resolution process can get more confusing. Which among the overloaded templates gets specialized depends on the order of declaration. Consider the code snippet:

```
//(a)
template <typename T> void F1(T t){}
//(b)
template <typename T> void F1(T* p){}
//(x): Specialization of template
template <> void F1<>(uint16_t* p){}
```

You cannot determine whether (x) specializes (a) or (b) from the declaration alone. it depends on the declaration order. For instance, in the preceding case (x) specializes (b). But in this case, (x) specializes (a):

```
//(a)
template <typename T> void F1(T t){}
//(x): Specialization of template
template <> void F1<>(uint16_t* p){}
//(b)
template <typename T> void F1(T* p){}
```

To avoid confusing code and unexpected behavior, avoid specializing function templates. If you must specialize a function template, then write a single function template that delegates to a class template. For example, in this code, a function template `f()` delegates to the class `f_implementation`.

```
template<class T> class f_implementation;

template<class T> void f( T t ) {
    FImpl<T>::f( t ); //Don't specialize function template
}

template<class T> class f_implementation {
    static void f( T t ); // Specializing class templates is permissible.
}
```

Delegating to a class template also enables partial specialization.

### Polyspace Implementation

If you explicitly specialize a function template, Polyspace flags the function template.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Specializing Function Templates

This example shows how Polyspace flags specialized function templates.

```
#include <cstdint>
#include <memory>
#include <iostream>
```

```
//(a)
template <typename T> void F1(T t){
  std::cout << "(a)" << std::endl;
}
//(x) specializes (a)
template <> void F1<>(uint16_t* p){// Noncompliant
  std::cout << "(x)" << std::endl;
}
//(b) overloads (a)
template <typename T> void F1(T* p){// Compliant
  std::cout << "(b)" << std::endl;
}
//(y) specializes (b)
template <> void F1<>(uint8_t* p){// Noncompliant
  std::cout << "(c)" << std::endl;
}
//(d) plain old function overloads (a) and (b)
void F1(uint8_t* p){                    // Compliant
  std::cout << "(d)" << std::endl;
}

int main(void)
{
    auto sp8 = std::make_unique<uint8_t>(3);
    auto sp16 = std::make_unique<uint16_t>(3);
    F1(sp8.get()); //calls (d), might expect (y)
    F1(sp16.get()); //calls (b), might expect (x)
    return 0;
}
```

When the function `F1()` is called in main, overload resolution determines which instance of `F1()` is called.

- When F1() is invoked with pointers to `uint8_t`, the compiler calls the plain-old-function (d) because it takes precedence. You might incorrectly expect the specialization (y) to be called.
- When F1() is invoked with pointers to `uint16_t`, the compiler calls the overloaded template (b) because it is the closest matching template. You might incorrectly expect the specialization (x) to be called.

Specializing function templates can cause confusion and unexpected results. Polyspace flags the specialized function templates (x) and (y).

## Check Information

**Group:** Templates
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-0-2

At least the basic guarantee for exception safety shall be provided for all operations. In addition, each function may offer either the strong guarantee or the nothrow guarantee

## Description

### Rule Definition

*At least the basic guarantee for exception safety shall be provided for all operations. In addition, each function may offer either the strong guarantee or the nothrow guarantee.*

### Rationale

To facilitate recovery from an exception while preserving the invariant of the relevant objects, programs must be designed to have exception safety. The C++ standard allows these levels of exception safety:

- Basic Exception Safety: This level of exception safety requires that after raising an exception, the basic invariants of all objects are maintained in a valid state and no memory is leaked. If an operation has basic exception safety, then you can destroy or assign to an object after the operations, even if an exception has been raised. The operations in standard library offers at least basic exception safety.
- Strong Exception Safety: This level of exception safety requires that after raising an exception, the state of the program remains as it was before the exception. Operations with strong exception safety either succeed or exit with an exception and have no effects on the program.
- `nothrow`: This level of safety requires that an exception cannot occur in an operation.

Without at least the basic exception safety, exceptions might corrupt the state of a program or create memory leaks. The AUTOSAR C++14 standard requires that a program must offer at least the basic exception safety for all operations, strong exception safety for key operations, and `nothrow` guarantee for some operations.

### Polyspace Implementation

Polyspace raises a violation of this rule if any of these conditions are true:

- An exception violates a class invariant: This condition is met if a non-`noexcept` member function attempts to raise an exception after modifying any of the fields in the class. For instance, Polyspace flags a `throw()` statement or a `new` statement if they are used after modifying the internal state of a class.
- An exception leaves dynamic memory in an invalid state: This condition is met if an exception is raised without deallocating the dynamic memory that is already allocated. For instance, Polyspace flags an exception raising statement if it is outside a `try` block or inside a `catch` block and the allocated resources are not deallocated before the statement. Exception raising statements might include:
  - A `throw` statement
  - Calls to functions containing `throw` statements
  - Calls to constructors containing `throw` statements.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Modifying Object Invariant Before Raising Exceptions

```
#include <cstdint>
#include <cstring>
template<typename T=int>
class myArray
{
public:
    myArray(){/*...*/}
    myArray(const myArray& rhs)
    {
        DeepCopyNC(rhs);
        DeepCopy(rhs);
    }
    ~myArray()
    {
        delete[] array;
    }
    void DeepCopyNC(const myArray& rhs) // Noncompliant
    {
        if (this != &rhs) {
            delete[] array;
            array = nullptr;
            len = rhs.len;

            if (len > 0) {

                array = new T[len];
                std::memcpy(array, rhs.array, len * sizeof(T));
            }
        }
    }
    void DeepCopy(const myArray& rhs) // Compliant
    {
        T* eTmp = nullptr;

        if (rhs.len > 0) {
            eTmp = new T[rhs.len];
            std::memcpy(eTmp, rhs.array, rhs.len * sizeof(T));
        }

        delete[] array;
        array = eTmp;
        len = rhs.len;
    }

private:
    T* array;
    std::size_t len;
};
```

```
extern     myArray<> c1{};
void foo(){

    myArray<> c2{c1};

}
```

This example shows a generic class template `myArray` that manages a raw array. The copy constructor of this class shows two implementations of deep copy. In the function `DeepCopyNC()`, the memory operations are performed after modifying the `this->array` and `this->len` fields. If the memory operation exits with an exception, the invariant of the original class is violated. Because the class invariant is violated by an exception, the function `DeepCopyNC()` does not have basic exception safety. Polyspace flags it. To provide basic exception safety, modify the class invariant only when the memory operation succeeds, as shown in `DeepCopy`.

### Deallocate Resource Before Raising Exception

```
#include<cstdlib>
#include<exception>
 class complex_ptr{

    complex_ptr(){
        real = (double*)malloc(sizeof(double));
        imag = (double*)malloc(sizeof(double));
        if(real==nullptr || imag==nullptr){
            throw std::exception(); //Noncompliant
        }
    }
    ~complex_ptr(){
        free(real);
        free(imag);
    }
    private:
    double* real;
    double* imag;

};
class complex_ptr2{

    complex_ptr2() {
        real = (double*)malloc(sizeof(double));
        imag = (double*)malloc(sizeof(double));
        if(real==nullptr || imag==nullptr){
            deallocate();
            throw std::exception(); //Compliant
        }
    }
    void deallocate(){
        free(real);
        free(imag);
    }
    ~complex_ptr2(){
        deallocate();
    }
    private:
    double* real;
    double* imag;
};
void foo(void){
    complex_ptr Z;
    complex_ptr2 X;
    //...
}
```

In this example, the class `complex_ptr` is responsible for allocating and deallocating two raw pointers to `double`. The constructor `complex_ptr::complex_ptr()` terminates with an exception when a memory allocation operation fails. The class goes out of scope before deallocating the already allocated resources, resulting in a memory leak. Because an exception causes a memory leak, this code does not have basic exception safety. Polyspace flags the `throw` statement in the constructor.

Similar to `complex_ptr`, the constructor of class `complex_ptr2` raises an exception when a memory allocation operation fails. Before raising the exception, the constructor deallocates the allocated memory by calling `deallocate()`. This constructor provides basic exception safety and is compliant with this rule.

### Handle Exceptions Arising from Object Construction

```
#include<exception>
extern int rd;

struct magnitude {
    magnitude() {
        //...
        throw std::exception();
    }
};


class complex_num {
public:
    complex_num() {
        theta = new int;
        r = new magnitude();//Noncompliant
    }
    magnitude* r;
    int* theta;
};

void foo() {
    complex_num t;
}
```

In this example, the constructor of `complex_num`constructs an `int` and a `magnitude` object. The constructor of a `magnitude` object might raise an exception, causing the already allocated `theta` to leak. Because an exception might lead to memory leak, this code does not have basic exception safety. Polyspace flags the call to `magnitude()`. To provide basic exception safety, construct the `magnitude` object within a `try-catch` block.

## Check Information

**Group:** Exception handling
**Category:** Required, Partially automated

## Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `AUTOSAR C++14 Rule A15-1-4` | `AUTOSAR C++14 Rule A15-2-2`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-0-3

Exception safety guarantee of a called function shall be considered

## Description

### Rule Definition

*Exception safety guarantee of a called function shall be considered.*

### Rationale

The exception safety level of a function affects the function behavior. Functions that have a basic exception safety do not modify the class invariant when they exit with an exception. Functions that have a strong exception safety restores the class invariant to the state it was before the function call.

If a function provides neither of these exception safety, calling it might lead to unexpected behavior. For instance, an external function might exit with an exception for a specific input. When a client uses this function, the exception might be unexpected, resulting in a memory leak or program termination.

### Polyspace Implementation

The `emplace()` method uses constructors of objects to construct them in-place within the containers. An exception arising from this method leaves the constructor in an invalid state. If the called constructors contain `throw()` statements, or calls a function that raises exception, Polyspace flags the `emplace()` method.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Using Emplace with Inputs That Raise Exceptions

```
#include<exception>
#include<vector>

class ratio{
public:
    ratio(double in1=1, double in2=1):num{in1}, denom{in2}{
        if(in2==0)
        throw std::invalid_argument("Denominator cannot be zero");

    }
private:
    double num;
    double denom;

};

void foo(){
    std::vector<ratio> v;
    v.reserve(10);
    std::vector<ratio>::const_iterator cit;
    //...
    v.emplace(cit, 1,0);//Noncompliant
}
```

In this example, creating a `ratio` with a zero denominator causes an exception. Because the `emplace()` method might leave the vector `v` in an invalid state, Polyspace flags the call to the `emplace()` method.

## Check Information

**Group:** Exception handling
**Category:** Required, Partially automated

# Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-0-7

Exception handling mechanism shall guarantee a deterministic worst-case time execution time

## Description

### Rule Definition

*Exception handling mechanism shall guarantee a deterministic worst-case time execution time.*

### Rationale

Compilers, such as GCC or CLang, use dynamic memory allocation in the implementations of their exception handling mechanism. Dynamic memory allocation during exception handling results in a nondeterministic worst-case execution time. Such exception handling functions might raise memory allocation errors during run time. Avoid dynamic memory allocation in exception handling functions. Implement the exception handling functions so that memory is allocated before the run time.

### Polyspace Implementation

Violations of this rule are raised when the exception handling functions in your code allocates or deallocates memory dynamically. Polyspace recognizes the commonly used dynamic memory allocation functions and exception handling functions, including:

- `__cxa_allocate_exception`
- `__cxa_throw`
- `__cxa_free_exception`
- `__cxa_begin_catch`
- `__cxa_end_catch`
- Stack unwinding functions, such as `_Unwind_RaiseException`, `_Unwind_Resume`, `_Unwind_DeleteException`.

To check user-defined exception handling implementations, specify the user-defined exception handling and dynamic memory managing functions by using the analysis option `-code-behavior-specifications`. In the code behavior specification XML file, specify a function as an exception handling function by using the behavior `EXCEPTION_HANDLING`. Specify a function as a dynamic memory managing function by using the behavior `MANAGES_MEMORY`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Dynamic Memory Allocation in Exception Handling Functions

```
#include <cstdlib>
extern "C" void * __cxa_allocate_exception(size_t thrown_size) {
  return new char[thrown_size];   //Noncompliant
}

char bigBuffer[10*1024];
char* bigBufferPtr = bigBuffer;

extern "C" void* __cxa_begin_catch(void* unwind_arg) {
  char* ptr = bigBufferPtr;       // Compliant

  bigBufferPtr += *(int*)unwind_arg;

  return (void*)ptr;
}
```

In this example, the operator `new` allocates dynamic memory in the exception handling function `__cxa_allocate_exception`. Polyspace flags the use of dynamic memory allocation in the exception handling function. The best practice is to allocate memory elsewhere and use the allocated memory during exception handling. For instance, in the exception handling function `__cxa_begin_catch`, there is no dynamic memory allocation. Instead, memory is allocated statically in `bigBuffer`, which is then used in the function. This method of memory allocation for exception handling purposes is compliant with this rule.

## Check Information

**Group:** Exception handling
**Category:** Required, Partially automated

# Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"
`AUTOSAR C++14 Rule A18-5-7`

# AUTOSAR C++14 Rule A15-1-1

Only instances of types derived from std::exception should be thrown

## Description

### Rule Definition

*Only instances of types derived from std::exception should be thrown.*

### Rationale

Raising generic objects as exceptions can make your code difficult to read and reuse. Consider this code where exceptions are raised in two different `try-catch` blocks.

```
try{
    //..
    throw 1; // 1 means logic error;
}
catch(...){
    //...
}
//...
try{
    //...
    throw  std::logic_error{"Logic Error"};
}
catch(std::exception& e){
    //..
}
```

In the first code block, the cause or the meaning of this exception is not clear. An ambiguous exception such as this one can make the code difficult to read and reuse. These type of generic exceptions might also clash with exceptions raised elsewhere in the code, making the exceptions more difficult to handle. This rule states that such generic objects are not acceptable as exception objects.

In the second code block, the meaning and cause of the exception is clearly communicated by raising a specific and unique type of object as an exception. Such `throw` statements also match standard conventions. Clearly communicating the developer intent, adhering to the standard conventions, and raising unique type of exceptions make the code easy to read, understand, and reuse.

The class `std::exception` provides a consistent interface to raise unique exceptions corresponding to specific errors. It is standard convention to use this interface for raising exceptions. To make your code readable and reusable, raise objects of specific types that are derived from `std::exception` as the exception. Generic objects of type `std::exception` cannot be unique. Such exceptions violate this rule.

### Polyspace Implementation

- Polyspace flags a `throw` statement if the type of the raised object is not a class that is publicly derived from `std::exception`.

- If the raised object is part of a multiple inheritance hierarchy, then Polyspace flags the object if none of the base classes derive publicly from `std::exception` or if the base classes do not include `std::exception`.
- If you use a `throw;` statement without an argument in a catch block, Polyspace does not flag the `throw;` statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Raise Exception by Using Classes That Are Publicly Derived from `std::exception`

This example shows how Polyspace flags `throw` statements that raise objects of a noncompliant class hierarchy.

```
#include <stdexcept>
#include <memory>

class ConventionalException : public std::logic_error{
public:
    using std::logic_error::logic_error;
};
class CustomException {};
class CustomException_derived : public CustomException {};
class PrivateDerived : private std::exception {};
class ProtectedDerived : protected std::exception {};
class MultipleInheritenceCompliant :
    public ConventionalException, public CustomException {
public:
    MultipleInheritenceCompliant()
    : ConventionalException("Logic Error and Data Error") {
    }
};
class MultipleInheritenceNoncompliant :
    public ProtectedDerived, public CustomException {};

void Foo() {
    throw std::exception();   // Noncompliant
    throw CustomException(); // Noncompliant
    throw CustomException_derived(); // Noncompliant
    throw PrivateDerived();       // Noncompliant
    throw ProtectedDerived();     // Noncompliant
    throw MultipleInheritenceCompliant(); // Compliant
    throw MultipleInheritenceNoncompliant(); // Noncompliant
    throw ConventionalException{"Logic Error"};//Compliant
    throw std::make_shared<std::exception> // Noncompliant
    (std::logic_error("Logic Error"));
}
```

- Polyspace flags the statement `throw std::exception()` because it raises a generic `std::exception` object, which might not be unique in your code.

- Polyspace flags the statement `throw CustomException()` because the class `CustomException` does not derive from `std::exception`. Polyspace flags the statement `throw CustomException_derived()` for the same reason.
- Polyspace flags the statement `throw PrivateDerived()` because the class `PrivateDerived` derives from `std::exception` privately. You cannot catch this exception by using a `catch(std::exception& e)` block because of the private inheritance between `std::exception` and `PrivateDerived`. Polyspace flags the statement `throw ProtectedDerived()` for the same reason.
- Polyspace flags the statement `throw MultipleInheritenceNoncompliant()` because the class `MultipleInheritenceNoncompliant` does not publicly derive from `std::exception`.
- Polyspace flags the statement `throw std::make_shared<std::exception>(std::logic_error("Logic Error"))` because this statement raises a `std::shared_ptr` object as an exception which does not derive from `std::exception`.
- Polyspace does not flag the statements `throw MultipleInheritenceCompliant()` and `throw ConventionalException{"Logic Error"}` because these statements raise objects of classes that are publicly derived from `std::exception`.

### Avoid `throw` Statements in Catch Blocks That Accept `std::exception` Objects

This example shows how Polyspace flags `throw` statements in a catch block.

```
#include <memory>
#include <stdexcept>
class MyException : public std::logic_error
{
public:
    using std::logic_error::logic_error;
    // Implementation
};
void catch_and_rethrow() {
    try {
        /* ... */
    }catch (std::exception e) {
        throw; // Compliant
        throw e; // Noncompliant
    }catch (std::exception& e) {
        throw e; // Noncompliant
        throw;   // Compliant
    }catch (MyException& Error) {
        throw Error; // Compliant
    }
}
```

- Empty `throw;` statements in catch blocks might raise different types of exceptions depending on the run-time context. Because Polyspace analyzes the `throw;` statements without any run-time context-specific information, it cannot determine what kind of object an empty `throw;` statement might raise. Polyspace does not flag `throw;` statements in catch blocks.
- Polyspace flags the statements `throw e` because they raise the `std::exception` object `e` as an exception. Avoid such `throw` statements in catch blocks that accept references to or instances of the `std::exception` class.
- Polyspace does not flag the statement `throw Error` because it raises an object of class `MyException` that derives from `std::exception`.

## Check Information

**Group:** Exception handling
**Category:** Advisory, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-1-2

An exception object shall not be a pointer

## Description

### Rule Definition

*An exception object shall not be a pointer.*

### Rationale

If your `throw` expression is a pointer to a dynamically allocated object, the deallocation point of the allocated resource becomes ambiguous. Such ambiguity might lead to a memory leak. Throwing pointers as exceptions might allow functions to access objects after their lifetime ends, which results in an undefined behavior.

Avoid using pointers as exceptions. Raise exceptions by copy instead.

### Polyspace Implementation

The checker raises a violation if a `throw` statement throws an exception of pointer type.

The checker does not raise a violation if a NULL pointer is thrown as exception. Throwing a NULL pointer is forbidden according to `AUTOSAR C++14 Rule M15-1-2`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Raising Pointers as Exceptions

```
extern int flag;
class A{/**/};
void foo(void){
    A a;
    A* a_pointer = new A;
    A& a_ref = a;
    if(flag==0){
        throw a;
    }

    else if(flag==2){
        throw a_pointer;//Noncompliant
    }

    else if(flag==-1){
        throw a_ref;
    }
    else if(flag==-2){
        throw &a; //Noncompliant
    }
}
```

In this example, the function `foo()` throws several exceptions. Polyspace flags the `throw` statements where the `throw` operand is a pointer. Raising exceptions by copy or by reference is compliant with this rule.

## Check Information

**Group:** Exception Handling
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-1-3

All thrown exceptions should be unique

## Description

### Rule Definition

*All thrown exceptions should be unique.*

### Rationale

If the same object is raised as exceptions in multiple places, handling these exceptions and debugging the code can be difficult. Raising unique exception objects simplifies the debugging process. Consider this code where multiple exceptions are raised.

```
void f1(){
    //...
    throw std::logic_error("Error");
}
void f2(){
    //...
    throw std::logic_error("Error");
}
void f3(){
    //...
    throw std::logic_error("f3: Unexpected Condition");
}
int main(){
    try{
        f1();
        f2();
        f3();
        catch(std::logic_error& e){
            std::cout << e.what() << '\n';
        }
    }
}
```

The functions `f1()` and `f2()` raise the same exception, while `f3()` raises a unique exception. During debugging, you cannot determine if an exception arises from `f1()` or `f2()`. You know when an exception arises from `f3()`. To make the debugging process simpler, raise unique exceptions. An exception is unique if either of these conditions is true:

- The exception type does not occur elsewhere in your project.
- The error message or the error code does not occur elsewhere in your project.

### Polyspace Implementation

Polyspace highlights `throw` statements that raise the same class, enum value, integer, or constant literal as exceptions, and flags the final throw statement that raise the same object. You might want to raise the same exception in multiple places by using a preconstructed exception object. Polyspace does not flag `throw` statements that raise such preconstructed exception objects. If you raise the same literal object in multiple places, Polyspace does not flag it if the literal is not a constant or if the literal is hidden behind a variable.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Raise Unique Exceptions

This example shows how Polyspace flags nonunique exceptions.

```
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>

int readLog();
enum ENUM {
    ENUM_VALUE_13 = 13,
    ENUM_VALUE_14 = 14,
};
const char* value_gen(int i) {
    if (i % 2 == 0) return "value_gen-0";
    else return "value_gen-1";
}
class CustomException : public std::invalid_argument {
public:
    CustomException() : std::invalid_argument(value_gen(readLog())) {}
};
int foo0(){
    //...
        throw std::logic_error("Invalid Logic"); // Compliant
    //...
        throw std::runtime_error("Runtime Error"); // Compliant
}
int foo1(){
    //...
        throw std::logic_error(value_gen(0));
    //...
        throw std::logic_error(value_gen(2));
}
int foo2(){
//..
        throw CustomException();
//..
        throw CustomException(); // Noncompliant
}

    int foo3(){
    const int localConstInt42 = 42;
    //..
        throw 42;
    //...
        throw localConstInt42; //Noncompliant
}
int foo4(){
    //..
        throw "RUNTIME_ERROR";
    //...
        throw "RUNTIME_ERROR"; // Noncompliant
}
int foo5(){
    //...
        throw ENUM_VALUE_14;
    //...
        throw ENUM_VALUE_14; // Noncompliant
}
```

- The function `foo0()` raises two different types of objects as exceptions. Polyspace does not flag the `throw` statements in `foo0()`.
- The function `foo1()` raises two exceptions that evaluate to be the same object during run-time. Because Polyspace analyzes the exception objects without runtime information, the exceptions appear unique and Polyspace does not flag the throw statements.
- The function `foo2()` raises two exceptions that evaluate to be different objects at run-time. Because Polyspace analyzes the exception objects without run-time information, the exceptions appear nonunique. Polyspace highlights the throw statements in `foo2()` and flags the final the throw statement.
- The function `foo3()` raises three exceptions by using objects that are the same at compile time. Polyspace highlights the throw statements in `foo3()` and flags the final the throw statement. For the same reason, the final `throw` statements in `foo4()` and `foo5()` are flagged.

## Check Information

**Group:** Exception handling
**Category:** Advisory, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-1-4

If a function exits with an exception, then before a throw, the function shall place all objects/ resources that the function constructed in valid states or it shall delete them.

## Description

### Rule Definition

*If a function exits with an exception, then before a throw, the function shall place all objects/ resources that the function constructed in valid states or it shall delete them.*

### Rationale

When a function exits with an exception, any resource or memory that the function allocated might not be properly deallocated. Consider this code:

```
FILE* FilePtr;
//...
void foo(){
    FilePtr = fopen("some_file.txt", "r");
//...
    if(/*error condition*/)
    throw ERROR_CODE;

    delete FilePtr;
}
```

The allocated file pointer is intended to be deallocated before the function finishes execution. But when an exception takes place, the function exits without deleting the pointer, which results in a memory leak. To avoid memory leaks, a function must set all resources that it allocates to a valid state before it goes out of scope. For instance, in the preceding code example, the function must delete the pointer `FilePtr` before the `throw` statement.

Instead of manually tracking the allocation and deallocation of resources, the best practice is to follow the "Resource Acquisition Is Initialization" (RAII) or the "Constructor Acquires, Destructor Releases" (CADre) design pattern. In this pattern, resource allocation is performed in constructors and resource deallocation is performed in destructors. The lifecycle of resources are controlled by scope-bound objects in this pattern. When functions reach the end of their scope, the acquired resources are properly released. Consider this code:

```
void releaseFile(std::FILE* fp) { std::fclose(fp); }
std::unique_ptr<std::FILE, decltype(&releaseFile)> FilePtr;
//...
void foo(){
    FilePtr(std::fopen("some_file.txt"),&releaseFile);
//...
    if(/*error condition*/)
    throw ERROR_CODE;
}
```

Here, the unique pointer `FilePTR` invokes the function `releaseFile` to delete the allocated resource once the function `foo` reaches the end of its scope, whether normally or because of an unhandled exception.

C++ smart pointers such as `std::unique_ptr` and `std::shared_ptr` follow the RAII pattern. They simplify managing the lifecycle of resources during exception handling. Avoid using raw pointers whenever possible.

### Polyspace Implementation

Polyspace flags an uncaught `throw` statement in a if the statement might result in resource leak. For instance,

- A `throw` statement outside a `try` block is flagged if the allocated resources are not deallocated before the statement.
- A `throw` statement in a `catch` block is flagged if the resources are not deallocated before raising the exception.

Polyspace does not flag a `throw` statement if it is within a `try` block that has an appropriate handler or if the exception is raised before allocating resources.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Deallocate Resources Before `throw` Statements

```
#include <cstdint>
#include <memory>
#include <stdexcept>
extern int sensorFlag() noexcept;
namespace Noncompliant{
    void func(){
        int* intPtr = new int;
        int data = sensorFlag();
        if(data==-1)//Error
        throw std::runtime_error("Unexpected value");//Noncompliant
        //...
        delete intPtr;
    }
}
namespace Compliant{
    void func(){
        int* intPtr = new int;
        int data = sensorFlag();
        if(data==-1){//Error
            delete intPtr;
            throw std::runtime_error("Unexpected value");//Compliant
        }
        //...
        delete intPtr;
    }
}
namespace BestPractice{
    void func(){
        std::unique_ptr<int> intPtr = std::make_unique<int>();
        int data = sensorFlag();
        if(data==-1){//Error
            throw std::runtime_error("Unexpected value");//Compliant
        }
        //...

    }
}
```

In this example, the function `Noncompliant::func()` manages the raw pointer `inPtr`. The function allocates memory for it, and then releases the memory after some operations. The function exits with an exception when `data` is `-1`. In this case, the function exits before releasing the allocated memory, resulting in a memory leak. To prevent memory leak, the allocated memory must be released before raising the exception, as shown in `Compliant::func`.

The best practice is to follow the RAII design pattern and use `unique_ptr` instead of a raw pointer. `BestPractice::func` shows an implementation of `func` that follows the RAII pattern. In this case, the memory lifecycle is managed by the object itself. That is, once `func` is out of scope, the smart pointer `intPtr` deletes itself and releases the memory. Because the memory management is performed correctly by the smart pointer, `BestPractice::func` is simpler and safer.

## Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-1-5

Exceptions shall not be thrown across execution boundaries

## Description

### Rule Definition

*Exceptions shall not be thrown across execution boundaries.*

### Rationale

An execution boundary separates code in your projects that are compiled by different compilers or different versions of a compiler. For example, you might use different compilers to compile your code and a library function. In this case, the execution boundary exists between the call site in your executable and the function implementation in the library.

Exception handling requires interoperability between the functions that raise the exception and the functions that handle the raised exceptions. Code on two sides of an execution boundary might implement exception handling by using incompatible interfaces. For instance:

```
// lib.h
void foo() noexcept(false);
//lib.cpp
void foo() noexcept(false){
  //...
  throw -1;
}
//App.cpp
#include"lib.h"
int main(){
  try{
    foo();
  }catch(int& e){
    //handle exception
  }
}
```

If you compile the library file `lib.cpp` by using GCC and compile the application `App.cpp` by using Microsoft Visual Studio, then these two portions of your code use incompatible exception-handling interfaces. The exception arising from the library is not caught, which terminates the application unexpectedly.

Avoid raising exceptions in code that you intend to reuse as a library in different projects. Instead of raising exceptions, return different error codes to handle unexpected situations.

As an exception, raising exceptions across an execution boundary is compliant with this rule if the codes on the two sides of the boundary are guaranteed to use the same exception handling interface

### Polyspace Implementation

Polyspace raises a violation of this rule if a library interface function raises an exception. Violations are not raised if a `new` operator in a such a function raises an `std::bad_alloc` exception.

To use this checker correctly, you must specify those functions in your code that are part of a library interface. Declare a function *`foo()`* as a library interface function by setting their `visibility` attribute. For instance:

- GNU and Clang compiler: `void __attribute__((visibility("default"))) foo(){/*...*/}`
- Visual Studio: `void __declspec(dllexport) foo(){/**.../}`

If you do not explicitly specify a function as visible, Polyspace assumes that it is not part of a library interface.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use `noexcept` Library Functions

```
#include<exception>
int FLAG;
void __attribute__((visibility("default"))) foo(int rd) noexcept(false) {//Noncompliant
    if (rd==-1)
        throw std::exception();
}

int __attribute__((visibility("default"))) bar(bool rd) noexcept(true) {
    if (rd==-1){
        FLAG = 52;
        return -1;
    }
}
```

In this example, the `visibility` attribute of the function `foo()` is set to `default`, which indicates that this function belongs in a library interface. Because `foo()` might be used in projects that have different exception handling mechanisms, exceptions raised by `foo()` might remain unhandled. Avoid raising exceptions from library interface functions. Instead, use other methods for handling errors, as shown in `bar`. The library function `bar` sets an error flag and returns an error code when a logic error occurs in the code.

# Version History

**Introduced in R2022b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A15-2-1

Constructors that are not noexcept shall not be invoked before program startup

## Description

### Rule Definition

*Constructors that are not noexcept shall not be invoked before program startup.*

### Rationale

In C++, the compiler responds to an exception by following these steps:

- The compiler tries to match the exception with a handler in the current scope or a higher scope.
- If the exception matches with a handler, then the handler accepts the exception and begins stack unwinding. During stack unwinding, The program execution moves from the scope that produces the exception to the outer scopes in reverse order. The program execution then invokes the destructors for each variable on the stack that are not destroyed yet. After stack unwinding, program execution resumes from the line immediately after the triggered handler.
- If the exception does not match a handler, then the compiler terminates the execution in an implementation-defined manner. That is, the exact process of program termination depends on the particular set of software and hardware that you use. For instance, the compiler might invoke `std::terminate()`, which in turn might invoke `std::abort()` to abnormally abort the execution. Based on the implementation, the stack might not be unwound before the program is aborted. If the stack is not unwound before program termination, then the destructors of the variables in the stack are not invoked, leading to resource leak and security vulnerabilities.

Before program startup, the constructors of static or global objects are invoked to construct and initialize these objects. If such a constructor raises an exception, the compiler might abnormally terminate the code execution without unwinding the stack. Consider this code where the constructor of the static object `obj` might cause an exception.

```
class A{
    A(){

        //...
    }
};

static A obj;

main(){
    //...
}
```

The static object `obj` is constructed by calling `A()` before `main()` starts. Because `A()` is called before program startup, no exception handler can be matched with exceptions raised by `A()`. Based on the implementation, such an exception can result in program termination without stack unwinding, leading to memory leak and security vulnerabilities.

Because exceptions raised by constructors of static or global objects cannot be matched to an exception handler, declare these constructors as `noexcept`.

### Polyspace Implementation

Polyspace flags statements where non-`noexcept` constructors of a static or global object are directly invoked. It also highlights the noncompliant constructors.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Declare Constructors of Static or Global Objects as `noexcept`

This example shows how Polyspace flags constructors of static or global objects.

```
#include <cstdint>
#include <stdexcept>
#include <string>
class A
{
public:
    A() noexcept : x(0){}
    A(std::int32_t n) : x(n) {
        throw std::runtime_error("Unexpected error");
    }
    A(std::int32_t i, std::int32_t j) noexcept : x(i + j)
    {
        try {
            throw std::runtime_error("Error");
        }
        catch (std::exception& e) {
        }
    }
private:
    std::int32_t x;
};

static A a1;     // Compliant
static A a2(5);  // Noncompliant
static A a6(5);  // Ignored because unused
static A a3(5, 10); // Compliant
A a4(5);  //Noncompliant
A a5(5, 10); //Compliant
int foo_A(A a) { };

int bar_A(int value) {
    A a{value};   //Compliant
    return foo_A(a);
}
int value2b = bar_A(20);   // Compliant
std::string s{"Hello World"};//Noncompliant
int value2a = foo_A(20);   //Noncompliant
int convert(){
    return  foo_A(a2);
}
```

- Polyspace flags the statement `std::string s{"Hello World"};` because this statement invokes the non-`noexcept` constructor of the string `s` before program startup.
- Polyspace flags the statement `A a4(5);` because this statement invokes the non-`noexcept` constructor `A(std::int32_t n)` before program startup.
- Polyspace flags the statement `static A a2(5);` because this statement invokes the non-`noexcept` constructor `A(std::int32_t n)` before program startup.
- Polyspace flags the statement `int value2a = foo_A(20);` because the implicit conversion from `int` to `A` requires invoking the non-`noexcept` constructor `A(std::int32_t n)` before startup.
- Polyspace does not flag the statement `static A a6(5);`. Because `a6` is not used in this code, the compiler does not construct the object. As a result, the non-`noexcept` constructor `A(std::int32_t n)` is not invoked before program startup.
- Polyspace does not flag the statement `A a{value};` in the body of the function `bar_A()` because the object `a` is local, and it is not created during program startup.

## Check Information

**Group:** Exception Handling
**Category:** Required, Automated

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-2-2

If a constructor is not noexcept and the constructor cannot finish object initialization, then it shall deallocate the object's resources and it shall throw an exception

## Description

### Rule Definition

*If a constructor is not noexcept and the constructor cannot finish object initialization, then it shall deallocate the object's resources and it shall throw an exception.*

### Rationale

When a constructor abruptly terminates due to unhandled exception or failed dynamic resource allocation, it might leave some objects in a partially constructed object, which is undefined behavior. Before raising exceptions in class constructors, deallocate the already allocated resources. When allocating resources, specify the `new` operation as `std::nothrow`. Alternatively, perform the resource allocation in a `try` or `function-try` block to handle exceptions that might arise from a failed allocation.

### Polyspace Implementation

Polyspace flags a `throw` or `new` statement outside a `try` block in a non-`noexcept` class constructor if the statement might result in resource leak. For instance:

- A `throw` statement outside a `try` block is flagged if the allocated resources are not deallocated before the statement.
- A `new` statement is flagged if there are more than one `new` statement in succession and the latter ones is not specified as `std::nothrow` or wrapped in a `try` or `function-try` block.

Polyspace ignores classes that remain unused in your code.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Deallocate Resource Before Raising Exception

```
#include<cstdlib>
#include<exception>
 class complex_ptr{

    complex_ptr(){
        real = (double*)malloc(sizeof(double));
        imag = (double*)malloc(sizeof(double));
        if(real==nullptr || imag==nullptr){
            throw std::exception(); //Noncompliant
        }
    }
    ~complex_ptr(){
        free(real);
        free(imag);
    }
    private:
    double* real;
    double* imag;

};
class complex_ptr2{

    complex_ptr2() {
        real = (double*)malloc(sizeof(double));
        imag = (double*)malloc(sizeof(double));
        if(real==nullptr || imag==nullptr){
            deallocate();
            throw std::exception(); //Compliant
        }
    }
    void deallocate(){
        free(real);
        free(imag);
    }
    ~complex_ptr2(){
        deallocate();
    }
    private:
    double* real;
    double* imag;

};
void foo(void){
    complex_ptr Z;
    complex_ptr2 X;
    //...
}
```

In this example, the class `complex_ptr` is responsible for allocating and deallocating two raw pointers to `double`. The constructor `complex_ptr::complex_ptr()` terminates with an exception when a memory allocation operation fails. The class goes out of scope before deallocating the already allocated resources, resulting in a partially constructed object. Polyspace flags the `throw` statement in the constructor.

Similar to `complex_ptr`, the constructor of class `complex_ptr2` raises an exception when a memory allocation operation fails. Before raising the exception, the constructor deallocates the allocated memory by calling `deallocate()`. This constructor is compliant with this rule.

#### Handle Exceptions Arising from `new` Operations in Constructors

```
#include<cstdlib>
#include <stdexcept>
#include <new>
 class complex_ptr{

    complex_ptr(): real(new double), imag(new double){ //Noncompliant
    }
    ~complex_ptr(){
        delete real;
        delete imag;
    }
    private:
    double* real;
    double* imag;

};

class complex_ptr2{

    complex_ptr2() try: real(new double), imag(new double){    //Compliant

    }catch(std::bad_alloc){
        //...
    }
    ~complex_ptr2(){
        delete real;
        delete imag;
    }
    private:
    double* real;
    double* imag;

};

void foo(void){
    complex_ptr Z;
    complex_ptr2 X;
    //...
}
```

In this example, the constructor of `complex_ptr` performs `new` operations that might raise exceptions. Because the constructor has no mechanism for handling these exceptions, they might cause the constructor to abruptly terminate. Such termination might leave the object in a partially defined state because the allocated resources are not deallocated. Polyspace flags the constructor. The constructor of `complex_ptr2` performs the new operations in a `function-try` block and handles potential exceptions in a `catch` block. This constructor is compliant with the rule because it handles the exceptions that might arise from the `new` operations.

## Check Information

**Group:** Exception handling
**Category:** Required, Partially automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-3-3

Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions

## Description

### Rule Definition

*Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions.*

### Rationale

During the execution of `main()` or a task main function, different exceptions can arise. For instance:

- Explicitly raised exceptions of class `std::exception`
- Exceptions arising from the third-party libraries that you use
- Unexpected exceptions

If any of these exceptions cannot be matched to a handler, the compiler implicitly invokes the function `std::terminate()` to abnormally terminate program execution. Depending on the hardware and software that you use, this termination process might invoke `std::abort()` to abort program execution without deleting the variables in the stack. Such an abnormal termination results in memory leaks and security vulnerabilities.

Unhandled exceptions might cause an abnormal termination of the program execution, leading to memory leaks and security vulnerabilities. To avoid these issues, execute the operations of `main()` or task main functions in a `try-catch` block. In the catch blocks:

- Handle exceptions of type `std::exception` explicitly in appropriate catch blocks.
- Handle the base class of exceptions arising from third-party libraries.
- Handle unexpected exceptions in a `catch(...)` block.

### Polyspace Implementation

- Polyspace flags a `main()` function or a task main function if :
  - Unhandled exceptions are raised in the function. For example, exceptions that are raised outside the `try-catch` block or in a catch block might remain unhandled.
  - The function does not have a `try-catch` block.
  - The function does not have catch blocks to explicitly handle `std::exception` type exceptions.
  - The function does not have a catch-all or `catch(...)` blocks to handle unexpected exceptions.
- Polyspace does not check if exceptions from third-party libraries are handled.
- Polyspace flags a `main()` function or a task main function even if the unhandled exception might not be raised.

Polyspace detects the `main()` function. To specify a function as a task main function, use these compilation options:

- -entry-points <*name*>
- -cyclic-tasks <*name*>
- -interrupts <*name*>

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Unhandled Exceptions in `main()` and Task Main Functions

This example shows how Polyspace flags `main()` and task main functions that do not handle all exceptions. To specify the functions `Noncompliant`, `Noncompliant2`, and `Compliant2` as task main functions, use the compile option `-entry-points Noncompliant,Noncompliant2,Compliant`.

```
#include <stdexcept>
void f_throw() {             // Compliant
  throw 1;
}
void Noncompliant()     // Noncompliant
{
  try {

  } catch (std::exception& e) {
      f_throw();             // throw
  } catch (...) {
      throw;
  }
}
int Noncompliant2()     // Noncompliant
{
  f_throw();              // throw
  try {

  } catch (std::exception& e) {
  } catch (...) {
  }
  return 0;
}
int Compliant()   // Compliant
{

  try {
    // program code
  } catch (std::runtime_error& e) {
    // Handle runtime errors
  } catch (std::logic_error& e) {
    // Handle logic errors
  } catch (std::exception& e) {
    // Handle all expected exceptions
  }catch (...) {
    // Handle all unexpected exceptions
  }
  return 0;
}
int main()   // Noncompliant
{

  try {
    // program code
  } catch (std::runtime_error& e) {
    // Handle runtime errors
  } catch (std::logic_error& e) {
    // Handle logic errors
  } catch (std::exception& e) {
    // Handle all expected exceptions
  }
  return 0;
}
```

- The function `f_throw()` exits with an unhandled exception. Because this function is not a main or task main function, Polyspace does not flag it.
- The function `Noncompliant()` and `Noncompliant2()` are specified as task main functions. In these functions, `f_throw` raises an exception that is not handled. Because these task main functions do not handle all the exceptions that might arise, Polyspace flags them. Enclose operations that might raise an exception in a `try-catch` block to handle exceptions that might arise.
- The function `main()` does not have a `catch(...)` block to handle any unexpected exceptions. Because the `main()` function does not handle all the exceptions that might arise, Polyspace flags it.
- The function `Compliant` is specified as a task main function. This function has a `catch` for all the exceptions that might arise. Polyspace does not flag it.

### Check Information

**Group:** Exception handling
**Category:** Required, Partially automated

## Version History

**Introduced in R2020b**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-3-4

Catch-all (ellipsis and std::exception) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines

## Description

### Rule Definition

*Catch-all (ellipsis and std::exception) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines.*

### Rationale

Catch-all handlers such as `catch(std::exception)` or `catch(...)` blocks match many different types of exceptions. If you handle an exception by using such a catch-all handler, you do not have detailed and specific information about the raised exception. Such catch-all handlers cannot take meaningful actions to handle the raised exceptions. These catch-all handlers are useful in processing unexpected exceptions by raising the exceptions again or by properly exiting from the application.

Because catch-all handlers are useful for specific purposes, it is inefficient to use them in every function. Use catch-all handlers in:

- Main functions
- Task main functions
- Functions that call a third-party function that might be noncompliant with AUTOSAR C++14 guidelines
- Functions that are designed to isolate independent components of your code

### Polyspace Implementation

Polyspace flags `catch(std::exception)` and `catch(...)` blocks in a function if none of these are true:

- The function is the `main()` function.
- The function is a task main function.
- The function calls an external or third-party function that might exit with an exception.

Polyspace detects the `main()` function. To specify a function as a task main function, use these compilation options:

- -entry-points <*name*>
- -cyclic-tasks <*name*>
- -interrupts <*name*>

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Catch-All Handlers in Task Main Functions

This example shows how Polyspace flags catch-all handlers in a task main function `EntryPoint` and a nonentry-point function `NonEntryPoint()`. To specify the function `EntryPoint` as a task main function, use the compile option `-entry-points EntryPoint`.

```
#include <stdexcept>
#define MYEXCEPTION std::exception &
class ExceptionBased: std::exception {
};
typedef std::exception MyException;
typedef std::exception & MyExceptionRef;

void NonEntryPoint()
{
    try {
        int i = 2;

        // ...
    } catch (int i) {                   // Compliant
    } catch (int &i) {                  // Compliant
    } catch (std::runtime_error e) {    // Compliant
    } catch (std::runtime_error& e) {   // Compliant
    } catch (std::exception *e) {       // Compliant
    } catch (std::exception e) {        // Noncompliant
    } catch (const std::exception& e) { // Noncompliant
    } catch(MyException e){             // Noncompliant
    } catch(ExceptionBased e){          // Compliant
    } catch (...) {                     // Noncompliant
    }
}
void EntryPoint() noexcept
{
    try {
        int i = 2;

        // ...
    } catch (MyException &e) {          // Compliant
    } catch (MyException e) {           // Compliant
    } catch (MyExceptionRef e) {        // Compliant
    } catch (ExceptionBased e) {        // Compliant
    } catch (const std::exception& e) { // Compliant
    } catch (MYEXCEPTION e) {           // Compliant
    }
}
```

The function `NonEntryPoint()` is not a `main()` or task main function. In this function, Polyspace flags these catch-all blocks:

- The `catch (std::exception e)` block matches the different types of exceptions that derive from the class `std::exception`. This catch-all handler is useful in a main or task main function. Because `NonEntryPoint()` is neither a `main()` nor a task main function, Polyspace flags the statement `catch (std::exception e)`. For the same reason, Polyspace flags the statement `catch (std::exception& e)`.

- `MyException` is a `typedef` of `std::exception`. The `catch(MyException e)` block matches the different types of exceptions that derive from the class `std::except`. Because `NonEntryPoint()` is neither a main nor a task main function, Polyspace flags the statement `catch(MyException e)`.
- Because `NonEntryPoint()` is neither a main nor a task main function, Polyspace flags the statement `catch(...)`

The function `EntryPoint()` is specified as a task main function. Polyspace does not flag the catch-all blocks in this function.

### Use Catch-All Handlers to Handle Exceptions Raised By Third-party Code

```
#include <stdexcept>
void Fextern_throw(void);
void Fextern_nothrow(void) noexcept(true);
void Foo0()
{
    try {
        Fextern_nothrow();
    } catch (...) {                 // Noncompliant
    }
}
void Foo1()
{
    try {
        try {
            Fextern_throw();
        } catch (...) {               // Compliant
        }
    } catch (std::exception& e) {   // Compliant
    }
}
void Foo2()
{
    try {
        try {
            Fextern_nothrow();
        } catch (...) {               // Noncompliant
            Fextern_throw();
        }
    } catch (std::exception& e) {   // Compliant
    }
}
```

- The function `Foo0()` calls the third-party function `Fextern_nothrow()`, which is specified as `noexcept(true)`. Because the third-party code is specified as `noexcept`, Polyspace flags the `catch(...)` block in `Foo0()`.
- The function `Foo1()` calls the third-party function `Fextern_throw()` that might raise an exception. Because the third-party code might raise an exception, Polyspace does not flag the catch-all handler blocks in `Foo1()`.
- The function `Foo2()` contains a nested `try-catch` block. In the inner block, the external function `Fextern_nothrow()` is called, which is specified as `noexcept(true)`. Polyspace flags the `catch(...)` block in the inner `try-catch` block. The catch-all block in the outer `try-catch` is compliant because this block handles the exceptions that might be raised by the external function `Fextern_throw()`.

## Check Information

**Group:** Exception handling
**Category:** Required, Non-automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A15-3-5

A class type exception shall be caught by reference or const reference

## Description

### Rule Definition

*A class type exception shall be caught by reference or const reference.*

### Rationale

If a class type exception is caught by value, the exception object might be sliced. For instance:

```
class baseException();
class derivedException : public baseException {};

void foo() {
    try {
        //...
        throw derivedException();
    }
    catch (baseException e) { //slices the thrown exception
        //...
    }
}
```

When the catch block in `foo()` catches the `derivedException` object, you might expect the object to remain a `derivedException` object. Because the object is caught by value, it is sliced to a `baseException` object. Unintended object slicing risks unexpected code behavior at run time. To avoid object slicing, catch class type exceptions by reference or `const` reference.

### Polyspace Implementation

Polyspace flags catch statements where class type exceptions are caught by value.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Catch Exceptions by Reference

```
#include <exception>
#include <iostream>

class baseException : public std::exception {
public:
    baseException() : exception() {}
    const char* what() const noexcept(true) override {
        return "Base Exception Object";
    }
};

class derivedException : public baseException {
public:
    derivedException() : baseException() {}
    const char* what() const noexcept(true) override {
        return "Derived Exception Object";
    }
};

class exampleException{};

void foo() {
    try {
        throw derivedException();
    }
    catch (baseException e) { //Noncompliant
        std::cout << e.what();
    }
    catch (derivedException e) { //Noncompliant
        std::cout << e.what();
    }
    catch (exampleException e) { //Noncompliant
    }
    catch (baseException &e) { //Compliant
        std::cout << e.what();
    }
    catch (const baseException &e) { //Compliant
        std::cout << e.what();
    }
    catch (derivedException &e) { //Compliant
        std::cout << e.what();
    }
    catch (const derivedException &e) { //Compliant
        std::cout << e.what();
    }
}
```

In this example, Polyspace flags the `catch` blocks that catches exception objects by value. For instance:

- Catch blocks for exceptions of type `baseException`, `derivedException`, and `exampleException` are noncompliant because the thrown class type exception is caught by value. These blocks might slice the exception objects.
- Catch blocks for exceptions of type `baseException &`, `const baseException &`, `derivedException &`, and `const derivedException &` are compliant because the class type exception is caught by reference or `const` reference.

## Check Information

**Group:** Exception Handling
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-4-1

Dynamic exception-specification shall not be used

## Description

### Rule Definition

*Dynamic exception-specification shall not be used.*

### Rationale

Dynamic exception specification is the method of specifying how a function behaves in case of an exception by using a `throw(`*`<list of exceptions>`*`)` statement in the function declaration. Using dynamic exception specification has these issues:

- Performance cost: Because dynamic exception specifications are checked at runtime, it adds to overhead and might reduce code performance.
- Not suitable for generic programming: Because the precise type of exceptions raised by function or class templates are generally not known beforehand, it can be difficult to use `throw` statements in generic code.

For these reasons, avoid the `throw(`*`<list of exceptions>`*`)` statement to specify exceptions. Use the `noexcept` keyword instead. Because the `noexcept` statements are checked at compile time, it is suitable for generic programming and generally provides better performance than `throw` statements. The C++11 standard specifies that dynamic exception specification will be removed from C++ in the future.

### Polyspace Implementation

When a `throw(`*`<list of exceptions>`*`)` statement is used in a function declaration, Polyspace flags the `throw` statement. Polyspace does not flag `throw` statements that are used for raising an exception.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using `throw()` for Dynamic Exception Specification

```
#include <string>
// throw for raising exception is compliant
void F9 () throw(std::runtime_error) {   //Non-compliant
    throw (std::runtime_error("foo"));   //Compliant
}
// Both declaration and definition is flagged
void F11 () throw(std::runtime_error);   //Noncompliant
void F11 () throw(std::runtime_error) {} //Noncompliant
// Instantiated and Noninstantiated templates are flagged
```

```
template <class T>
void F10 () throw(std::runtime_error) {  //Noncompliant
    throw (std::runtime_error("foo"));   //Compliant
}

template <class T>
void F12 () throw(std::runtime_error);   //Noncompliant
template <class T>
void foo() noexcept(noexcept(T())) {}//Compliant

void bar () {
    foo<int>();  // noexcept(noexcept(int())) => noexcept(true)
    F10<std::string> ();                 //Compliant

}
```

Polyspace flags statements such as `throw(std::runtime_error)` that are used in the function declarations and definitions as dynamic exception specification. Avoid dynamic exception specification. Use the keyword `noexcept` instead. The template `foo` uses the `noexcept` keyword as the exception specification, which is compliant with this rule.

## Check Information

**Group:** Exception handling
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A15-4-2

If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception

## Description

### Rule Definition

*If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception.*

### Rationale

You can specify that a callable entity does not raise an exception by specifying it as `noexcept`, or `noexcept(true)`, or `noexcept(<true condition>)`. The compiler expects that a `noexcept` function does not exit with an exception. Based on this assumption, the compiler omits the exception handing process for `noexcept` functions. When a `noexcept` function exits with an exception, the exception becomes unhandled.

If a `noexcept` function exits with an exception, the compiler invokes `std::terminate()` implicitly. The function `std::terminate()` terminates the program execution in an implementation-defined manner. That is, the exact process of program termination depends on the particular set of software and hardware that you use. For instance, `std:terminate()` might invoke `std::abort()` to abnormally abort the execution without unwinding the stack, leading to resource leak and security vulnerabilities.

Specify functions as `noexcept` or `noexcept(true)` only when you know the functions raise no exceptions. If you cannot determine the exception specification of a function, specify it by using `noexcept(false)`.

### Polyspace Implementation

If you specify a callable entity by using `noexcept`, `noexcept(true)`, or `noexcept(<true condition>)`, Polyspace checks the callable entity for unhandled exceptions and flags the callable entity if it might exit with an exception.

When a callable entity invokes other callable entities, Polyspace makes certain assumptions to calculate whether there might be unhandled exceptions.

- Functions: When a `noexcept` function calls another function, Polyspace checks whether the called function might raise an exception only if it is specified as `noexcept(<false>)`. If the called function is specified as `noexcept`, Polyspace assumes that it does not raise an exception. Some standard library functions, such as the constructor of `std::string`, use pointers to functions to perform memory allocation, which might raise exceptions. Because these functions are not specified as `noexcept(<false>)`, Polyspace does not flag a function that calls these standard library functions.
- External function: When a `noexcept` function calls an external function, Polyspace flags the function declaration if the external function is specified as `noexcept(<false>)`.
- Virtual function: When a function calls a virtual function, Polyspace flags the function declaration if the virtual function is specified as `noexcept(<false>)` in a derived class. For instance, if a

noexcept function calls a virtual function that is declared as noexcept(<true>) in the base class, and noexcept(<false>) in a subsequent derived class, Polyspace flags the declaration of the noexcept function.

- Pointers to function: When a noexcept function invokes a pointer to a function, Polyspace assumes that the pointer to function does not raise exceptions.

When analyzing whether a function raises unhandled exceptions, Polyspace ignores:

- Exceptions raised in destructors
- Exceptions raised in atexit() operations

Polyspace also ignores the dynamic context when checking for exceptions. For instance, a function might raise unhandled exceptions that arise only in certain dynamic contexts. Polyspace flags such a function even if the exception might not be raised.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Handle Possible Exceptions Within noexcept Functions

This example shows how Polyspace flags noexcept functions that might raise unhandled exceptions. Consider this code containing several noexcept functions. These functions invoke other callable entities like functions, external functions, and virtual functions.

```
#include <stdexcept>
#include <typeinfo>
void LibraryFunc();
void LibraryFunc_noexcept_false() noexcept(false);
void LibraryFunc_noexcept_true() noexcept(true);



void SpecFalseCT() noexcept  // Noncompliant
{
    try {
        LibraryFunc_noexcept_false();
    } catch (int &e) {
        LibraryFunc_noexcept_false();
    } catch (std::exception &e) {
    } catch (...) {
    }
}

class A {
public:
    virtual void f() {}
};

class B : A {
public:
    virtual void f() noexcept {}
};

class C : B {
public:
    virtual void f() noexcept {}
};

class D : A {
public:
    virtual void f() noexcept(false) { throw(2);}
};

void A1(A &a) noexcept {          // Noncompliant
    a.f();
}

void D2(D &d) noexcept {          //Compliant
    try {
        d.f();
    } catch (int i) {
    } catch (...) {
    }
}

void B2(B *b) noexcept {          // Compliant
    b->f();
}
template <class T>
T f_tp(T a) noexcept(sizeof(T)<=4)    // Noncompliant
{
    if (sizeof(T) >4 ) {
        throw std::runtime_error("invalid case");
    }
    return a;
}
void instantiate(void)
{
    f_tp<char>(1);
}
void f() noexcept {               //Noncompliant
    throw std::runtime_error("dead code");
}

void g() noexcept {               // Compliant
    f();
}
```

- Polyspace flags the declaration of the function template `f_tp` because:
  - The condition `sizeof(T)<=4` evaluates to `true` for `char` so the template becomes a `noexcept(true)` function.
  - Polyspace analyzes the `noexcept(true)` instance of the template statically. Polyspace deduces that the template might raise an exception because of the `throw` statement, even though the condition `sizeof(T)>4` is `false`. That is, Polyspace flags the template even though the `throw` statement is never reached.

  Polyspace ignores function templates that are not instantiated.

- Polyspace flags the noexcept function SpecFaleCT() because this function calls the noexcept(false) external function LibraryFunc_noexcept_false() without encapsulating it in a try-catch block. Any exceptions raised by this call to the external function might raise an unhandled exception.
- Polyspace flags the declaration of the noexcept function A1() because this function might call the noexcept(false) function D.f() when the input parameter a is of class D. Depending on the class of the input parameter, the noexcept polymorphic function A1() might raise an unhandled exception.
- Polyspace flags the function f() because it is a noexcept function that uses throw to raise an unhandled exception. Polyspace does not flag the noexcept function g() even though it calls f() because f() is specified as noexcept.
- Polyspace does not flag the noexcept function D2() even though it calls the noexcept(false) function D.f() because D2() handles the exceptions that might arise by using a catch(...) block.

## Check Information

**Group:** Exception Handling
**Category:** Required, Automated

## See Also

Check AUTOSAR C++ 14 (-autosar-cpp14)

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-4-3

The noexcept specification of a function shall either be identical across all translation units, or identical or more restrictive between a virtual member function and an overrider

## Description

### Rule Definition

*The noexcept specification of a function shall either be identical across all translation units, or identical or more restrictive between a virtual member function and an overrider.*

### Rationale

Translation units are the different source files that the compiler compiles. When a function has a different exception specification in different source files, it might result in undefined behavior. Similarly, a different exception specification of a polymorphic function in different levels of a class hierarchy might result in compilation failure in some cases. Depending on the software and hardware that you use, different exception specifications of a function in different places might cause a compilation failure or result in undefined behavior leading to security vulnerabilities.

To avoid undefined behavior and security vulnerabilities:

- Keep the same exception specification in all declarations of a function.
- If a virtual function is declared by using `noexcept` or `noexcept(true)` as the exception specification, declare the overrider functions in the derived classes by using the same specification.
- If a virtual function is declared by using `noexcept(false)` as the exception specification, declare the overrider functions in the derived classes by using either `noexcept(false)` or `noexcept(true)` as the exception specification.

### Polyspace Implementation

Polyspace flags the exception specification of a function if the function is declared with different exception specifications in different places in a file. Polyspace flags an overrider function in a derived class if it is specified as `noexcept(fale)` while the virtual function in the base class is specified as `noexcept`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Maintain Identical Exception Specification in Function Declarations

This example shows how Polyspace flags declarations of one function that has different exception specifications. In one file `file1.cpp`, the member functions of the classes `A` and `B` are declared.

```
//file1.cpp
```

```
class A
{
public:
    void F() noexcept;
    void G() noexcept(false);

};
class B
{
public:
    void W() noexcept;
    void R() noexcept(false);

};
```

In another file `file2.cpp`, the member functions of these classes are defined.

```
// file2.cpp
#include"file1.cpp"

void A::F() noexcept(false) //Noncompliant
{
    // Implementation
}
void A::G() noexcept //Noncompliant
{
    // Implementation
}


void B::W() noexcept //Compliant
{
    // Implementation
}
void B::R() noexcept(false) //Compliant
{
    // Implementation
}
```

To see the violations of this rule, run Polyspace and specify both `file1.cpp` and `file2.cpp` as source files by using the option `-sources`. Keep `file1.cpp` and `file2.cpp` in the same folder. For more details about specifying multiple source files, see `-sources`.

The compilation might fail, but Polyspace flags the functions with nonidentical exception specification.

- The function `A::F()` is declared in`file1.cpp` by using the exception specification `noexcept`, but it is declared by using the exception specification `noexcept(false)` in `file2.cpp`. Polyspace flags the nonidentical exception specification in the latter declaration. For the same reason, the exception specification of `A::G()` in `file2.cpp` is also flagged.
- The functions `B::W()` and `B::R()` are declared and defined by using the same exception specification in the two source files. These functions are compliant with this rule.

### Keep Exception Specification Identical or More Restrictive When Declaring Overrider Functions

This example shows how Polyspace flags the declaration of overrider functions that have less restrictive exception specifications.

```
class A
{
public:
    virtual void V1() noexcept = 0;
    virtual void V2() noexcept(false) = 0;
    virtual void V3() noexcept = 0;
};
class B : public A
{
public:
    void V1() noexcept(false) override //Noncompliant
    {
        // Implementation
    }
    void V2() noexcept override //Compliant
    {
        // Implementation
    }
    void V3() noexcept override //Compliant
    {
        // Implementation
    }
};
```

The pure virtual functions `A::V1()`, `A::V2()`, and `A::V3()` are implemented by the overriding functions `B::V1()`, `B::V2()`, and `B::V3()` respectively.

- Polyspace flags the function `B::V1()` because this overriding function is specified by using the less restrictive exception specification `noexcept(false)` compared to the base class virtual function `A::V1()`, which is specified by using `noexcept`.
- Polyspace does not flag `B::V2()` because this overriding function is specified by using the more restrictive specification `noexcept` compared to the base class virtual function `A::V2()`, which is specified by using `noexcept(false)`.
- Polyspace does not flag `B::V3()` because this overriding function is specified by using the same exception specification as the base class virtual function `A::V3()`.

## Check Information

**Group:** Exception handling
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A15-4-4

A declaration of non-throwing function shall contain noexcept specification

## Description

### Rule Definition

*A declaration of non-throwing function shall contain noexcept specification.*

### Rationale

Specifying functions that do not raise exceptions by using the specifier `noexcept` or `noexcept(true)` enables the compiler to perform certain optimizations for these functions, such as omitting the exception handling process. Specifying the exception specification of functions clearly communicates that you expect the functions to not raise exceptions.

Specify functions that do not raise exceptions by using the specifier `noexcept`. If the exception specification of a function depends on a template argument, use `noexcept(<condition>)`. If the exception specification of a function is unknown, assume it raises exceptions.

### Polyspace Implementation

Polyspace flags the definition of a callable entity, such as function, class or function template, or class constructors if the following is true:

- The callable entity is defined. Polyspace does not flag functions that are declared but not defined. Polyspace checks function or class templates that have at least one instantiation.
- The callable entity raises no exceptions. In case of templates of classes and functions, at least one instantiation raises no exception. When checking callable entities for exceptions, Polyspace assumes external functions with no definitions behave as `noexcept(true)`. For more information about how Polyspace checks if a callable entity raises an exception, see the Polyspace Implementation section of `AUTOSAR C++14 Rule A15-4-2`.
- The callable entity has no exception specification.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Specify Nonthrowing Functions as `noexcept`

```
#include <iostream>
#include <stdexcept>
void F1();
void F2() noexcept;
void F3() noexcept(true);
void F4() noexcept(false);

void F5(){ //Noncompliant
```

```
    F2();
    F3();

}

void F6() noexcept    // Compliant
{
    try {
        F4();
    }catch (std::exception& e) {
        // Handle exceptions
    }
}

class NotThrowing {
public:
    NotThrowing() { // Noncompliant
        F2();
    }
};

class Throwing{
public:
    Throwing() { // Compliant
        F4();
    }
};
template <class T, bool B> void CompliantClasshandler() noexcept(B) { //Compliant
    //...
}

template <class T> void NoncompliantClasshandler() { // Noncompliant
    //...
}

int Factory() noexcept{
    Throwing a;
    NotThrowing b;
    NoncompliantClasshandler<Throwing>();
    NoncompliantClasshandler<NotThrowing>();
    CompliantClasshandler<Throwing, true>();
    CompliantClasshandler<NotThrowing, false>();
    return 1;
}
```

- The function `F5` does not raise exceptions but it is not marked as `noexcept` in the code. Polyspace flags the definitions of the function. The function `F6` does not raise exceptions and it is specified as `noexcept` in the code. Polyspace does not flag `F6`.
- Polyspace checks constructors of classes when the classes are used in the code. In this example, the classes `Throwing` and `NotThrowing` are instantiated and Polyspace checks their constructors.

  The constructor of the class `NotThrowing` does not raise exceptions but it is not specified as `noexcept` in the code. Polyspace flags the function. The constructor of the class `Throwing` raises exception and it is not specified as `noexcept` in the code. Polyspace does not flag this constructor.

- Polyspace checks function templates when the template is instantiated. In this example, Polyspace checks the templates `CompliantClasshandler` and `NoncompliantClasshandler` because they are instantiated in `Factory`.

  Because the instantiation of `NoncompliantClasshandler` in `Factory` with the class `NotThrowing` does not raise exception, and the template is not specified as `noexcept` in the code, Polyspace flags the definition of `NoncompliantClasshandler`. Polyspace does not flag the template `CompliantClasshandler` because it is specified by a conditionalized `noexcept` operator in the code.

## Check Information

**Group:** Exception handling
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-4-5

Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders

## Description

### Rule Definition

*Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders.*

### Rationale

In C++, there are no checked exceptions because the compiler does not force functions to specify or handle the exceptions that the functions might raise. Dynamic exception specification of the form `throw(<>)` is obsolete and error-prone. The exception specification prescribed in the C++ standard specifies only whether a function raises an exception or not by using the specifier `noexcept`. Because there is no official way to declare which exceptions might arise from a function, the AUTOSAR standard requires that each function declaration be accompanied by comments that document the exception handling of the function. This method of documenting the exceptions is similar to the JAVA exception handling mechanism.

Use comments to specify a list of exceptions that a static analysis tool must check. Before function declarations, use comments to document which of the checked exceptions are expected in the function.

For a class template, the possible exceptions depend on the template argument. Because you cannot predicts the possible exceptions arising from a class template, this rule does not apply for templates.

### Polyspace Implementation

Polyspace reports a violation of this rule when any of these conditions are true:

- A function raises a checked exception but does not document it before its declaration.
- A function does not raise all the checked exceptions that are documented in comments before its declaration.
- A function documents an unchecked exception.
- A function documents an exception but does not define it.

This checker ignores the class member functions that are not called in your code. Because this rule does not apply for templates, Polyspace does not report violations of this rule on templates.

When documenting the checked exception classes, separate different checked exception classes by using line breaks, except between the class declaration and the documenting comments. For example, in this code, a line break separates the checked exceptions class `A` and `B`. The documenting comment and declaration of each class are kept together:

```
/// @checkedException
class A{};
```

```
/// @checkedException
class B{};
```

#### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

#### Document Checked Exceptions by Using Comments

```
#include <cstdint>
#include <stdexcept>

class ObjectType1{};
class ObjectType2{};

/// @checkedException
class TypeError : public std::exception
{
    // Implementation
};

/// @checkedException
class DimMismatch : public std::exception
{
    // Implementation
};

/// @checkedException
class SizeError : public std::exception
{
    // Implementation
};

/// @throw TypeError Unexpetced Object Type as Input
/// @throw DimMismatch Container Dimension Mismatched
/// @throw SizeError    Object Size Too large
void Transform1(ObjectType1& substrate,
               ObjectType2& coating) noexcept(false) //Compliant
{
    // ...
    throw TypeError();
    // ...
    throw DimMismatch();
    // ...
    throw SizeError();
    // ...
}

/// @throw TypeError Unexpetced Object Type as Input
void Transform2(ObjectType1& substrate,//Noncompliant
             ObjectType2& coating) noexcept(false)
//The function raises SizeError
//but does not document it before its declaration.
```

```
{
    // ...
    throw TypeError();
    // ...
    throw SizeError();
    // ...
}

class ValidationError : std::exception

{
    // Implementation
};

/// @throw TypeError Unexpetced Object Type as Input
/// @throw DimMismatch Container Dimension Mismatched
/// @throw SizeError   Object Size Too large
/// @throw ValidationError Checksum is Negative
void Transform3(ObjectType1& substrate,//Noncompliant
               ObjectType2& coating) noexcept(false)
// The function does not raise all the @throw exceptions
// The function documents an unchecked exception ValidationError.
{
    // ...
    throw TypeError();
    // ...
    throw SizeError();
    // ...
}

/// @throw TypeError Unexpetced Object Type as Input
/// @throw LengthMismatch Array Length Mismatched
void Transform4(void){//Noncompliant
// The function does not define the documented exception LengthMismatch
    // ...
    throw TypeError();
    // ...

}
```

In this example, the functions list their checked exceptions at the beginning of the file by using the tag `@checkedException` in comments. These functions then specify which of these checked exceptions are raised in their bodies by using the comment tag `@throw` before their declarations.

- The function `Transform1` is compliant with this rule because it specifies three checked exceptions before its declaration and then raises the same checked exceptions in its body.
- The function `Transform2` is not compliant with this rule because it raises the checked exception`SizeError` but does not document the exception in the comments before the function declaration.
- The function `Transform2` violates the rule in two different ways:
  - The function documents the checked exception `DimMismatch` in the comments before the function declaration but does not raise it in the function body.
  - The function documents the exception `ValidationError` before its declaration, but the exception is not listed as a checked exception.

- The function `Transform4` is noncompliant because it documents an exception `LengthMismatch` but the code does not have the definition of this type.

### Check Information

**Group:** Exception handling
**Category:** Required, Automated

## Version History

**Introduced in R2021a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-5-1

All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate

## Description

### Rule Definition

*All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate.*

### Rationale

This rule states that certain functions must not exit with an exception.

- Destructors and deallocation functions: When an exception is raised, the compiler invokes the destructors and deallocation functions to safely delete the objects in the stack. If a destructor or a deallocation function exits with an exception at that time, the compiler terminates the program execution abnormally. Depending on the software or hardware that you use, abnormal program termination can result in resource leaks and security vulnerabilities. To prevent these issues, avoid destructors and deallocator functions that might exit with an exception. Default destructors and deallocators are `noexcept` functions. When you provide a custom destructor or deallocation function, specify them as `noexcept` and handle all exceptions within the function so that they do not exit with exceptions. For a polymorphic class hierarchy, this rule applies to the destructors of the base and all derived classes.
- Move constructors and move assignment operators: If a move constructor or a move assignment operator exits with an exception, it cannot be guaranteed that the program will revert to the state it was before the move operation. Avoid a move constructor or a move assignment operator that might exit with an exception. Specify these functions as `noexcept` because standard library functions might avoid move operations unless they are declared as `noexcept`. You can also declare these special member functions as `=default`. For more information on when you can declare the special member functions as `=default`, see `AUTOSAR C++14 Rule A12-0-1`.
- Swap functions: Developers expect that a swap function does not exit with an exception. If a swap function exits with an exception, standard library algorithms and copy operations might not work in your code as expected. Specify swap functions as `noexcept`. Avoid operations that might exit with an exception in swap functions.

When you use templates as generic move constructors, generic move assignment operators, and generic swap functions, these templates can have dynamic exception specifications without violating this rule.

### Polyspace Implementation

Polyspace flags a user-defined destructor, deallocation function, move constructor, move assignment operator, and swap function if it might raise an exception. If a function is named `swap` or `Swap` and takes a reference as input, Polyspace considers it a swap function.

Polyspace ignores functions that are declared but not defined.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Exceptions in Destructors and Deallocation Functions

This example shows how Polyspace flags destructors and deallocation functions that might raise an exception. Consider this code with two classes.

```
#include <stdexcept>
class Compliant
{
public:
    //...
    ~Compliant()  //Compliant
    {
        try {
            // ...
            throw std::runtime_error("Error");
        }
        catch (std::exception& e) {
        //...
        }
    }
};

class Noncompliant
{
public:
    //...
    ~Noncompliant()
    {
        throw std::runtime_error("Error"); //Noncompliant
    }
    static void operator delete(void* ptr, std::size_t sz)
    {
        // ...
        throw std::runtime_error("Error");    // Noncompliant
    }
};
```

- The destructor of `Compliant` raises an exception by using a `throw` statement. Because this exception is handled within the destructor function by using a `try-catch` block, `~Compliant()` is compliant with this rule.
- The destructor of `Noncompliant` also raises an exception by using a `throw` statement. Because this exception is not handled within the function, the destructor `~Noncompliant()` exits with an exception. Polyspace flags this `throw` statement in the destructor.
- The deallocation function `Noncompliant::delete()` does not comply with this rule because it does not handle the exception raised within the function. Polyspace flags the `throw` statement in the function.

### Avoid Exceptions in Move Operations

Polyspace flags move operators or move constructors if:

- They might exit with an exception
- They are not specified as `noexcept`

Consider this code where move operations are implemented for two classes.

```
#include <stdexcept>
class Compliant
{
    //...
public:
    Compliant(Compliant&& rhs) noexcept   //Compliant
    {
        try {
            // ...
            throw std::runtime_error("Error");
        }
        catch (std::exception& e) {
            //...
        }
    }

    Compliant& operator=(Compliant&& rhs) noexcept //Compliant
    {
        try {
            // ...
            throw std::runtime_error("Error");
        }
        catch (std::exception& e) {
            //...
        }
        return *this;
    }
};

class Noncompliant
{
public:
    //...
    Noncompliant(Noncompliant&& rhs) //Noncompliant
    {
        // ...
        throw std::runtime_error("Error");    //Noncompliant
    }
    Noncompliant& operator=(Noncompliant&& rhs) //Noncompliant
    {
        // ...
        throw std::runtime_error("Error");   //Noncompliant
        return *this;
    }

};
```

- The move assignment operator and move constructor of the class `Compliant` are specified as `noexcept` and these functions handle exceptions that arise within them. The move constructor and move assignment operator of `Compliant` are compliant with this rule.
- The move assignment operator and the move constructor of the class `Noncompliant` are not specified as `noexcept`. Polyspace flags the declaration of these functions.
- The move assignment operator and the move constructor of the class `Noncompliant` contain `throw` statement that raise exceptions without handling them within these functions. Polyspace flags these `throw` statements.

### Avoid Exceptions in Swap Functions

Consider this code containing two swap functions.

```
#include <stdexcept>
namespace Compliant{
    class C1{};
    void Swap(C1& lhs, C1& rhs) noexcept   //Compliant
    {
        // Implementation
    }
}
namespace Noncompliant{
    class C2{};
    void Swap( C2& lhs, C2& rhs ) noexcept(false) //Noncompliant
    {
        throw std::runtime_error( "Error" );   //Noncompliant
    }
}
```

- The function `Compliant::Swap()` is specified as `noexcept` and does not raise an exception. This swap function is compliant with this rule.
- The function `Noncompliant::Swap()` is specified as `noexcept(false)` and it exits with an exception. Polyspace flags the exception specification of the function and the `throw` statement.

## Check Information

**Group:** Exception handling
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A15-5-2

Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done

## Description

### Rule Definition

*Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done.*

### Rationale

Functions such as `std::abort()`, `std::quick_exit()`, and `std::_Exit()` terminate the program immediately without invoking any exit handlers or calling any destructors for the constructed objects. The `std::terminate()` function implicitly calls `std::abort()` to terminate the program abruptly. Exceptions that are unhandled or cannot be handled might also cause abrupt termination of the program.

Depending on your environment, the compiler might not release the allocated resources and unwind the stack when the program is terminated abruptly, leading to issues such as memory leaks. Such abnormal program terminations might make the code vulnerable to denial-of-service attacks. Avoid terminating the program abruptly.

### Polyspace Implementation

Polyspace flags the operations that might result in abrupt termination of the program. For instance:

- The destructor of a class exits with an unhandled exception. See `AUTOSAR C++14 Rule A15-5-3`.
- The constructor of a global or a static object is invoked directly but it is not explicitly specified as `noexcept`. See `AUTOSAR C++14 Rule A15-2-1`.
- A `noexcept` function raises an unhandled exception. See `AUTOSAR C++14 Rule A15-4-2`.
- The argument of a `throw` statement raises an exception. See `AUTOSAR C++14 Rule M15-1-1`.
- Unsafe termination functions such as `std::_Exit`, `std::abort`, and `std::quick_exit` are explicitly invoked.
- The function `std::terminate` is explicitly invoked.
- A handler for abnormal termination is explicitly registered by using the functions `std::set_terminate` or `std::get_terminate`.
- A handler for normal termination that is registered to `std::atexit` raises an unhandled exception.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Handle Possible Exceptions Within noexcept Functions

This example shows how Polyspace flags `noexcept` functions that might raise unhandled exceptions. Consider this code containing several `noexcept` functions. These functions invoke other callable entities such as functions, external functions, and virtual functions.

```
#include <stdexcept>
#include <typeinfo>
void LibraryFunc();
void LibraryFunc_noexcept_false() noexcept(false);
void LibraryFunc_noexcept_true() noexcept(true);



void SpecFalseCT() noexcept  // Noncompliant
{
    try {
        LibraryFunc_noexcept_false();
    } catch (int &e) {
        LibraryFunc_noexcept_false();
    } catch (std::exception &e) {
    } catch (...) {
    }
}

class A {
public:
    virtual void f() {}
};

class B : A {
public:
    virtual void f() noexcept {}
};

class C : B {
public:
    virtual void f() noexcept {}
};

class D : A {
public:
    virtual void f() noexcept(false) { throw(2);}
};

void A1(A &a) noexcept {          // Noncompliant
    a.f();
}

void D2(D &d) noexcept {          //Compliant
    try {
        d.f();
    } catch (int i) {
    } catch (...) {
    }
}

void B2(B *b) noexcept {          // Compliant
    b->f();
}
template <class T>
T f_tp(T a) noexcept(sizeof(T)<=4)    // Noncompliant
{
    if (sizeof(T) >4 ) {
        throw std::runtime_error("invalid case");
    }
    return a;
}
void instantiate(void)
{
    f_tp<char>(1);
}
void f() noexcept {               //Noncompliant
    throw std::runtime_error("dead code");
}

void g() noexcept {               // Compliant
    f();
}
```

- Polyspace flags the declaration of the function template `f_tp` because:
  - The condition `sizeof(T)<=4` evaluates to `true` for `char` so the template becomes a `noexcept(true)` function.
  - Polyspace analyzes the `noexcept(true)` instance of the template statically. Polyspace deduces that the template might raise an exception because of the `throw` statement, even though the condition `sizeof(T)>4` is `false`. That is, Polyspace flags the template even though the `throw` statement is never reached.

  Polyspace ignores function templates that are not instantiated.
- Polyspace flags the `noexcept` function `SpecFaleCT()` because this function calls the `noexcept(false)` external function `LibraryFunc_noexcept_false()` without encapsulating it in a `try-catch` block. Any exceptions raised by this call to the external function might raise an unhandled exception.
- Polyspace flags the declaration of the `noexcept` function `A1()` because this function might call the `noexcept(false)` function `D.f()` when the input parameter `a` is of class `D`. Depending on the class of the input parameter, the `noexcept` polymorphic function `A1()` might raise an unhandled exception.
- Polyspace flags the function `f()` because it is a `noexcept` function that uses `throw` to raise an unhandled exception. Polyspace does not flag the `noexcept` function `g()` even though it calls `f()` because `f()` is specified as `noexcept`.
- Polyspace does not flag the `noexcept` function `D2()` even though it calls the `noexcept(false)` function `D.f()` because `D2()` handles the exceptions that might arise by using a `catch(...)` block.

### Avoid Expressions That Can Raise Exceptions in `throw` Statements

This example shows how Polyspace flags the expressions in `throw` statements that can raise unexpected exceptions.

```
int f_throw() noexcept(false);

class WithDynamicAlloc {
public:
    WithDynamicAlloc(int n) {
        m_data = new int[n];
    }
    ~WithDynamicAlloc() {
        delete[] m_data;
    }
private:
    int* m_data;
};

class MightThrow {
public:
    MightThrow(bool b) {
        if (b) {
            throw 42;
        }
    }
};

class Base {
    virtual void bar() =0;
};
class Derived: public Base {
    void bar();
};
class UsingDerived {
public:
    UsingDerived(const Base& b) {
        m_d =
        dynamic_cast<const Derived&>(b);
    }
private:
    Derived m_d;
};
class CopyThrows {
public:
    CopyThrows() noexcept(true);
    CopyThrows(const CopyThrows& other) noexcept(false);
};
int foo(){
    try{
        //...
        throw WithDynamicAlloc(10); //Noncompliant
        //...
        throw MightThrow(false);//Noncompliant
        throw MightThrow(true);//Noncompliant
        //...
        Derived d;
        throw  UsingDerived(d);// Noncompliant
        //...
        throw f_throw(); //Noncompliant
        CopyThrows except;
        throw except;//Noncompliant
    }
    catch(WithDynamicAlloc& e){
        //...
    }
    catch(MightThrow& e){
        //...
    }
    catch(UsingDerived& e){
        //...
    }
}
```

- When constructing a `WithDyamicAlloc` object by calling the constructor `WithDynamicAlloc(10)`, exceptions can be raised during dynamic memory allocation. Because the expression `WithDynamicAlloc(10)` can raise an exception, Polyspace flags the `throw` statement `throw WithDynamicAlloc(10);`
- When constructing a `UsingDerived` object by calling the constructor `UsingDervide()`, exceptions can be raised during the dynamic casting operation. Because the expression `UsingDerived(d)` can raise exceptions, Polyspace flags the statement `throw UsingDerived(d)`.
- In the function `MightThrow()`, exceptions can be raised depending on the input to the function. Because Polyspace analyzes functions statically, it assumes that the function `MightThrow()` can raise exceptions. Polyspace flags the statements `throw MightThrow(false)` and `throw MightThrow(true)`.
- In the statement `throw except`, the object `except` is copied by implicitly calling the copy constructor of the class `CopyThrows`. Because the copy constructor is specified as `noexcept(false)`, Polyspace assumes that the copy operation might raise exceptions. Polyspace flags the statement `throw except`.
- Because the function `f_throw()` is specified as `noexcept(false)`, Polyspace assumes that it can raise exceptions. Polyspace flags the statement `throw f_throw()`.

### Avoid Unsafe Termination Functions

```
#include<cstdlib>
class obj
{
public:
    obj() noexcept(false){}
    obj(const obj& a){/*...*/}
    ~obj(){/*...*/}
};
static obj staticObject;
void foo(){
    obj localObject;
    //...
    std::_Exit(-1);//Noncompliant
}
void bar(){
    obj localObject;
    //...
    std::abort;//Noncompliant
}
void foobar(){
    obj localObject;
    //...
    std::quick_exit(-1);//Noncompliant
}
```

In this example, unsafe termination functions are invoked to terminate the program. These functions do not perform the essential cleanup operations such as calling destructors. For instance, the destructors of the `staticObject` or the three instances of `localObject` are not invoked. Any resource allocated by these objects is leaked. Polyspace flags the use of such unsafe termination programs.

### Avoid Explicitly Calling `std::terminate`

```
#include<cstdlib>
#include<exception>
class obj
{
public:
    obj() noexcept(false){}
    obj(const obj& a){/*...*/}
    ~obj(){/*...*/}
};
static obj staticObject;

void foobar(){
    obj localObject;
    //...
    std::terminate();//Noncompliant
}
```

Invoking `std::terminate` explicitly might result in abrupt termination of the program without calling the destructors of the local and static objects. Polyspace flags the explicit calls to `std::terminate`.

### Avoid Unhandled Exceptions in Termination Handlers

```
#include <stdexcept>
void atexit_handler(){//Noncompliant
    throw std::runtime_error("Error in atexit function");
}
void main(){
    try{
        //...
        std::atexit(atexit_handler);
    }catch(...){

    }
}
```

The termination handler `atexit_handler` raises an uncaught exception. The function `atexit_handler` executes after the main finishes execution. Unhandled exceptions in this function cannot be handled elsewhere, leading to an implicit call to `std::terminate()`. Polyspace flags the function.

### Avoid Unhandled Exceptions in Constructors and Destructors

```
#include <stdexcept>
#include <new>
class obj
```

```
{
public:
    obj() noexcept(false){}
    obj(const obj& a){
        //...
        throw -1;
    }
    ~obj()
    {
        try{
            // ...
            throw std::runtime_error("Error2"); // Noncompliant
        }catch(std::bad_alloc& e){

        }
    }
};
obj globalObject; //Noncompliant
```

In this example, the constructor of the object `globalObject` is specified as `noexcept`. The destructor of the object explicitly raises an unhandled exception. These unhandled exception might arise before the execution starts or after the `main` function completes execution, which might result in an abrupt and unsafe termination by invoking `std::abort()`. Polyspace flags these operations.

## Check Information

**Group:** Exception handling
**Category:** Required, Partially automated

# Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

`AUTOSAR C++14 Rule A15-5-3`
`AUTOSAR C++14 Rule A15-2-1`
`AUTOSAR C++14 Rule A15-4-2`
`AUTOSAR C++14 Rule M15-1-1`
"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A15-5-3

The std::terminate() function shall not be called implicitly

## Description

### Rule Definition

*The std::terminate() function shall not be called implicitly.*

### Polyspace Implementation

The checker flags situations that might result in calling the function `std::terminate()` implicitly. These situations might include:

- An exception remains unhandled. For instance:
  - While handling an exception, it escapes through another function that raises an unhandled exception. For instance, a catch statement or exception handler invokes another function that raises an unhandled exception.
  - An empty `throw` statement raises an unhandled exception again.

  For more details, see `Uncaught exception`
- A class destructor raises an exception.
- A termination handler that is passed to `std::atexit` raises an unhandled exception.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Unhandled Exceptions

```
#include <stdexcept>
#include <new>
class obj
{
public:
    obj() noexcept(false){}
    obj(const obj& a){
        //...
        throw -1;
    }
    ~obj()
    {
        try{
            // ...
            throw std::runtime_error("Error2"); // Noncompliant
        }catch(std::bad_alloc& e){
        }
    }
};
obj globalObject;
void atexit_handler(){//Noncompliant
    throw std::runtime_error("Error in atexit function");
}
void main(){
    try{
        //...
        obj localObject = globalObject; //Noncompliant
        std::atexit(atexit_handler);
    }catch(std::exception& e){

    }
}
```

In this example, Polyspace flags unhandled exceptions because they result in implicit calls to `std::terminate()`.

- The destructor `~obj()` does not catch the exception raised by the `throw` statement. The unhandled exception in the destructor results in abrupt termination of the program through an implicit call to `std::terminate`. Polyspace flags the `throw` statement in the destructor of `obj`.
- The `main()` function does not handle all exceptions raised in the code. For instance, the `main` function has no `catch` blocks to handle the exception raised by the copy constructor of `obj`. This unhandled exception might result in an implicit call to `std::terminate()`. Polyspace reports a violation on the statement where the copy constructor is invoked.
- The termination handler `atexit_handler` raises an uncaught exception. The function `atexit_handler` executes after the main finishes execution. Unhandled exceptions in this function cannot be handled elsewhere, leading to an implicit call to `std::terminate()`. Polyspace flags the function.

## Check Information

**Group:** Exception Handling
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A16-0-1

The preprocessor shall only be used for unconditional and conditional file inclusion and include guards, and using specific directives

## Description

### Rule Definition

*The preprocessor shall only be used for unconditional and conditional file inclusion and include guards, and using specific directives.*

### Rationale

Other than unconditional and conditional file inclusion and include guards, avoid the use of preprocessor directives. Use a safer alternative instead. For instance:

- Instead of:

  ```
  #define MIN(a,b) ((a < b)? (a) : (b))
  ```

  You can use inline functions and function templates.
- Instead of:

  ```
  #define MAX_ARRAY_SIZE 1024U
  ```

  You can use a constant object.

In these situations, preprocessor directives do not provide the benefits that the alternatives provide, such as linkage, type checking, overloading, and so on.

### Polyspace Implementation

The rule checker does not allow the use of preprocessor directives. The only exceptions are:

- `#ifdef`, `#ifndef`, `#if`, `#if defined`, `#elif`, `#else` and `#endif`, only if used for conditional file inclusion and include guards.
- `#define` only if used for defining macros to be used in include guards. For instance, in this example, the macro `__FILE_H__` prevents the contents of the header file from being included more than once:

  ```
  /* aHeader.h */

  #ifndef __FILE_H__
  #define __FILE_H__
     /* Contents of header file */
  #endif
  ```

  When `#ifdef`, `#define` and `#endif` are used as include guards in a header file, the entire content of the header file must be in the include guard.
- `#include`

The checker does not allow the `#define` directives in other contexts. If you use `#define`-s for purposes other than for include guards, do one of the following:

- To define macros when compiling your code, instead of `#define`-s, use compilation flags (such as the GCC option `-D`). When running a Polyspace analysis, use the equivalent Polyspace option `Preprocessor definitions (-D)`.
- To retain the use of `#define` in your code, justify the violation using comments in your results or code. See "Address Results in Polyspace User Interface Through Bug Fixes or Justifications".

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Compliant and Noncompliant Use of Preprocessor Directives

```
#include <cstdint>       //Compliant: unconditional file inclusion

#ifdef WIN32             //Compliant: include guard
   #include <windows.h>  //Compliant: conditional file inclusion
#endif

#ifdef WIN32     //Noncompliant
    std::int32_t func(std::int16_t x, std::int16_t y) noexcept;
#endif
```

In this example, the rule is not violated when preprocessor directives are used for unconditional and conditional inclusion and include guards. Otherwise, the rule is violated.

## Check Information

**Group:** Preprocessing directives
**Category:** Required, Automated

# Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A16-2-1

The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive

## Description

### Rule Definition

*The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A16-2-2

There shall be no unused include directives

## Description

### Rule Definition

*There shall be no unused include directives.*

### Rationale

Unused include directives unnecessarily increase the code base size and introduce dependencies, which slows down the compilation process. To avoid this issue, remove unused include directives.

Remove unused include directives from the code.

### Polyspace Implementation

Polyspace reports a violation when a source file includes a header file that contains declarations of symbols that are not used in the source file. This rule checker does not check any external libraries.

Polyspace considers an include directive to be unused when you do not:

- Invoke macros defined in the included file or the included file's includes.
- Use data types defined in the included file or the included file's includes.
- Call or use references to an object or function defined in the included file or the included file's includes.
- Use an externally visible object provided by the included file outside of the file.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unused Include Directive Used In Source File

Because the source file does not use a symbol, macro, or data type defined in the `<algorithm>` header file, Polyspace reports a violation on the `<algorithm>` header file. Remove the unused include directive from the source file.

```
#include <algorithm>    //Noncompliant
#include <cstdint>
#include <iostream>


void main()
{
    int32_t num = 5;
    std::cout << "Hello world : " << num;
}
```

## Check Information

**Group:** Preprocessing directives
**Category:** Required, Automated

# Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A16-2-3

An include directive shall be added explicitly for every symbol used in a file

## Description

### Rule Definition

*An include directive shall be added explicitly for every symbol used in a file.*

### Rationale

To avoid a compilation failure, declare every symbol, macro, and data type before use and include their associated header files in the source file.

### Polyspace Implementation

Polyspace reports a violation when a source file contains a symbol, macro, or data type and the header file in which the symbol, macro, or data type is defined in is not included in the same source file.

Polyspace does not report a violation if the header file containing the required include directive shares a name with the source file.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Include Directive Not Explicitly Added

The source code in this example contains the type `int32_t`. Because the source code does not include the header file where `int32_t` is defined, `<cstdint>`, Polyspace flags this violation. This example results in a compilation failure.

```
#include <ostream>
#include <iostream>
#include <string_view>

void main()
{
    int32_t num = 5;    //Noncompliant
    std::cout << "Hello world : " << num;
}
```

## Check Information

**Group:** Preprocessing directives
**Category:** Required, Non-automated

## Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A16-6-1

#error directive shall not be used

## Description

### Rule Definition

*#error directive shall not be used.*

### Rationale

You typically use the `#error` directive by combining it with a `#if` or similar directive to make the compilation fail and issue a message when a condition is not met. However, you cannot apply `#error` to templates. Preprocessor directives do not obey linkage, type checker, overloading and other C++ features, and `#error` will not be evaluated as a per-instance template deduction.

Instead, use `static_assert` for compile-time error checking. Static assertions provide all the benefits of C++ features and make the code clearer.

### Polyspace Implementation

Polyspace flags all uses of the `#error` directive.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Preprocessing directives
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A16-7-1

The #pragma directive shall not be used

## Description

### Rule Definition

*The #pragma directive shall not be used.*

### Rationale

The use of the `#pragma` directive in your code results in implementation-defined behavior. The directive might also not be supported by certain compilers.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `#pragma once` Directive

```
//header.h
#pragma once //Noncompliant

#ifndef HEADER_H_ //Compliant
#define HEADER_H_
// ...
// body of header file
//..
#endif
```

The `#pragma once` directive prevents the inclusion of `header.h` more than once. However, if you copy `header.h` into multiple project modules, the directive may or may not treat the copies as the same file depending on the implementation. To avoid double definitions, use the `#ifndef` include guard instead.

## Check Information

**Group:** 16 Preprocessing Directives
**Category:** Required, Automated

# Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A17-0-1

Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined

## Description

### Rule Definition

*Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined.*

### Rationale

Defining, redefining, or undefining reserved identifiers, macros, and functions that are in the C++ standard library is not recommended. Undefined behavior might occur when defining, redefining, or undefining reserved words such as:

- `__LINE__`
- `__FILE__`
- `__DATE__`
- `__TIME__`
- `__STDC__`
- `errno`
- `assert`

### Polyspace Implementation

Polyspace flags the preprocessor directives that define, redefine, or undefine reserved C++ standard library identifiers, macros, and functions.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Defining, Redefining, and Undefining Reserved C++ Identifiers, Macros, and Functions

```
#define __TIME__ 3  //Noncompliant
#undef __TIME__   //Noncompliant

#define __cplusplus 3 //Noncompliant
#undef __cplusplus //Noncompliant

#define break 3 //Noncompliant
#undef break //Noncompliant
```

```
#define pow 3 //Noncompliant
#undef pow //Noncompliant


#define example 3 //Compliant
#undef example //Compliant

#define example 7 //Compliant
```

In this example, Polyspace flags preprocessor directives that define or undefine reserved C++ identifiers and macros. For instance:

- Defining and undefining `__TIME__` is noncompliant because `__TIME__` is a reserved C++ macro.
- Defining and undefining `__cplusplus` is noncompliant because `__cplusplus` is a reserved C++ macro.
- Defining and undefining `break` is noncompliant because `break` is a reserved C++ identifier.
- Defining and undefining `pow` is noncompliant because `pow` is a reserved C++ function.
- Defining, undefining, and redefining `example` is compliant because `example` is not a reserved C++ macro, identifier, or function.

## Check Information

**Group:** Library Introduction
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A17-1-1

Use of the C Standard Library shall be encapsulated and isolated

## Description

### Rule Definition

*Use of the C Standard Library shall be encapsulated and isolated.*

### Rationale

The C Standard Library functions leave the responsibility for handling errors, data races and security issues to developers. For instance, some C Standard Library functions return specific values on errors. A developer calling one of those functions has to explicitly check its return value for those errors.

If all calls to C Standard Library functions are isolated and encapsulated in wrapper functions, the wrapper functions can be left to perform the checking. Callers of those wrapper functions are freed from the responsibility of handling specific error values from the C Standard Library function calls. Instead, the callers can handle errors from the wrapper function calls like any other exception.

### Polyspace Implementation

The checker flags functions that contain a call to a C Standard Library function and one of these C++-specific constructs:

- A call to a C++ Standard Library function.
- A `try` block.
- A `new` or `delete` operator.
- Range-based `for` loops.

These C++-specific constructs indicate that the call to the C Standard Library function is not fully encapsulated and isolated. The event list below a rule violation shows the locations of the C Standard Library function call and the C++-specific construct.

Note that when C Standard Library functions are redeclared in the `std` namespace, their specifications remain unchanged. The rationale for encapsulating continues to apply to `std::` versions of the C library functions. Therefore, the checker flags these versions, too.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### C Standard Library Function Call Not Isolated and Encapsulated

```
#include <cerrno>
#include <cstdio>
```

```
#include <cstring>
#include <iostream>
#include <stdexcept>

void func1() { //Compliant
    FILE* fptr;
    fptr = fopen("myfile.txt", "w");
    // C-style error handling
    if (fptr == NULL) {
        throw std::system_error(errno, std::system_category());
    }

    fclose(fptr);
}

void func1_caller() noexcept{
    // C++-style error handling
    try {
        func1();
    }
    catch(std::system_error& e) {
        std::cerr << "Error "<< e.code() << ": " << e.what();
    }
}

std::int32_t func2() { //Noncompliant
    FILE* fptr;
    fptr = fopen("myfile.txt", "w");
    // C-style error handling
    if (fptr == NULL) {
        std::cerr << "Error " << strerror(errno);
        return errno;
    }

    // C++-style error handling
    try {
        fclose(fptr);
    }
    catch (std::exception& e) {
        fclose(fptr);
    }
    return errno;
}
```

In this example, the call to the C Standard Library function `fopen()` is encapsulated in the wrapper function `func1()`. The wrapper function performs all the error handling required after the call. Functions calling `func1()`, for instance, `func1_caller()`, do not have to concern themselves about the `fopen()` errors and can handle exceptions from `func1()` like other exceptions.

Conversely, the function `func2()` does not fully encapsulate the `fopen()` call and returns the value of `errno`. Note that `fopen()` sets the value of `errno` after errors. Callers of `func2()` have to read the `errno` value via the `func2()` return and if the value indicates an error, perform any resource cleanup required.

The use of the `try` block in `func2()` indicates that the `fopen()` call is not fully isolated and encapsulated.

## Check Information

**Group:** Library introduction
**Category:** Required, Non-automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A17-6-1

Non-standard entities shall not be added to standard namespaces

## Description

### Rule Definition

*Non-standard entities shall not be added to standard namespaces.*

### Rationale

Adding declarations or definitions to namespace `std` or its subspaces:or to `posix` or its subspaces, leads to undefined behavior. For instance, any addition within braces here leads to undefined behavior:

```
namespace std {
  ...
}
```

Likewise, explicitly specializing a member function or member class of a standard library leads to undefined behavior.

### Polyspace Implementation

The checker flags additions to the namespaces `std`, `posix`, or their subspaces, or specializations of class or function templates from these namespaces.

The rule specification allows exceptions to the specialization aspect of the rule for standard library templates that require a user-defined type. If you have a process that all rule violations must be justified and an issue flagged by the checker belongs to this category of exceptions, justify the issue using comments in your result or code. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Library introduction
**Category:** Required, Automated

## Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-0-1

The C library facilities shall only be accessed through C++ library headers

## Description

### Rule Definition

*The C library facilities shall only be accessed through C++ library headers.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Language Support Library
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-0-2

The error state of a conversion from string to a numeric value shall be checked

## Description

### Rule Definition

*The error state of a conversion from string to a numeric value shall be checked.*

### Rationale

Converting strings to a numeric value might result in error conditions, for instance, when the input string:

- Does not contain a number
- Contains a number, but is out of range
- Contains additional data after a number

If you use C standard library functions such as `atoi()`, the preceding input errors might result in undefined behavior. To avoid undefined behavior and undetected errors, check the error state of output when converting strings to a numeric value. Avoid using C standard library function. Use C++ standard library functions, such as `std::stoi()`, `std::stof()`, and `std::stol()` instead.

### Polyspace Implementation

Polyspace flags the C standard library string-to-number functions of `atoi()`, `atol()`, and `atof()`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use C++ Library Functions for Converting Strings to Numeric Value

```
#include <cstdlib>
#include <iostream>
#include <string>

void foo() {
    std::string str1 = "7";
    std::string str2 = "3.1415";
    std::string str3 = "three";

    int myint1 = std::stoi(str1); //Compliant
    float myint2 = std::stof(str2); //Compliant
    long myint3 = std::stol(str3); //Compliant

    const char* str4 = "12";
    const char* str5 = "2.7182";
    const char* str6 = "undefinedError";
    int num4 = atoi(str4); //Noncompliant
    float num5 = atof(str5); //Noncompliant
    long num6 = atol(str6); //Noncompliant
    //...
}
```

In this example, Polyspace flags the use of C standard library functions for converting strings to numeric value. For instance:

- The string-to-number functions from the C standard library, such as `atoi()`, `atof()`, and `atol()` are noncompliant flagged because an invalid conversion results in undefined behavior.
- The string-to-number functions from the C++ standard library `std::stoi()`, `std::stof()`, and `std::stol()` are not flagged because an invalid conversion produces a `std::invalid_argument` exception, which is defined behavior.

## Check Information

**Group:** Language Support Library
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-0-3

The library <clocale> (locale.h) and the setlocale function shall not be used

## Description

### Rule Definition

*The library <clocale> (locale.h) and the setlocale function shall not be used.*

### Polyspace Implementation

`setlocale` and `localeconv` should not be used as a macro or a global with external "C" linkage.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Support Library
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A18-1-1

C-style arrays shall not be used

## Description

### Rule Definition

*C-style arrays shall not be used.*

### Rationale

A C-style array is an array that is not wrapped in a class such as `std::array` when the array is declared. You can lose information about the size of a C-style array. For instance, an array that you pass to a function decays to a pointer to the first element of the array. This can lead to unsafe and difficult to maintain code.

The AUTOSAR standard allows declarations of `static constexpr` data members of a C-style array type. For example, this declaration is compliant.

```
class A
{
  public:
    static constexpr std::uint8_t array[] {0, 1, 2}; // Compliant by exception
};
```

### Polyspace Implementation

The rule checker does not flag C-style array arguments in function declarations because the rule violation still exists if you fix the function declaration and not the definition. A function might be declared in your code and defined in a library that you cannot access. The checker flags C-style array arguments in function definitions. For instance, in this code snippet, the checker flags the argument of `foo` but not the argument of `bar`.

```
extern void bar(char arg[]); //Declaration, checker raises no rule violation
int foo(char arg[]) // Definition, checker raises a rule violation
{
    return sizeof(arg); //Returns size of pointer, not size of array
}
void baz()
{
    char value[10]; //C-style array, checker raises a rule violation
    assert(sizeof(value) == foo(value));
}
```

The checker raises a flag on `arg` in the definition of `foo` even when there is no explicit C-style array definition for the argument. For example, declaring `char* value;` instead of `char value[10];` in `baz()` would still result in a rule violation on the argument of `foo`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Declaration of C-Style Array

```
#include <array>

void func()
{

    const std::uint8_t size = 10;
    std::int32_t a1[size]; //non-compliant
    std::array<std::int32_t, size> a2; //compliant

}
```

In this example, the rule is violated when you declare C-style array `a1`. To declare fixed-size stack-allocated arrays, use `std:array` instead.

## Check Information

**Group:** 18 Language Support Library
**Category:** Required, Automated

# Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-1-2

The std::vector<bool> specialization shall not be used

## Description

### Rule Definition

*The std::vector<bool> specialization shall not be used.*

### Rationale

The specialization of `std::vector` for the type `bool` can be made space-efficient in an implementation defined manner. For instance, `std::vector<bool>` does not necessarily store its elements as a contiguous array. As a result, the specialization does not work as expected with all standard library template (STL) algorithms, such as the index `operator[]()` which does not return a contiguous sequence of elements. You cannot safely modify distinct elements of STL container `std::vector<bool>`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Non-Compliant and Compliant Use of `std::vector` With `bool` Type

```
#include <cstdint>
#include <vector>

class BoolWrapper
{
public:
    BoolWrapper() = default;
    constexpr BoolWrapper(bool b) : b_(b) {}
    constexpr operator bool() const
    {
        return b_;
    }
private:
    bool b_{};
};

void Fn() noexcept
{
    std::vector<bool> v2; //non-compliant
    std::vector<BoolWrapper> v3{true, false, true, false}; //compliant
}
```

In this example, vector `v2` is non-compliant because it is declared with `std::vector<bool>`. A possible fix is to use `std::vector` with a value type `BoolWrapper` that wraps `bool`.

## Check Information

**Group:** 18 Language Support Library
**Category:** Required, Automated

## Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-1-3

The std::auto_ptr shall not be used

## Description

### Rule Definition

*The std::auto_ptr shall not be used.*

### Rationale

The `std::auto_ptr` is a type of class template that predates the introduction of move semantics in the C++11 language standard. When you copy a source `std::auto_ptr` object into a target object, the source object is modified. The compiler transfers the ownership of the resources in the source object to the target object and sets the source object to a null-pointer. Because of this unusual copy syntax, using the source object after the copy operation might lead to unexpected behavior. Consider this code snippet where the use of `std::auto_ptr` results in a segmentation fault.

```
void func(auto_ptr<int> p) {
    cout<<*p;
    //...
}

int main()
{
    std::auto_ptr<int> s = new int(1);
    //..
    func(s); // This call makes s a null-pointer
    //...
    func(s); // exception, because s is null
    return 1;
}
```

The first call to `func()` copies the source `std::auto_ptr` object `s` to the argument `p`, transfers ownership of the pointer to `p`, and sets `s` to a null pointer. When `func()` is called again, the compiler tries to access the null-pointer `s`, causing a segmentation fault.

The `std::auto_ptr` type objects are also incompatible with any generic code that expects a copy operation to not invalidate the source object, such as the standard template library (STL). Avoid using `std::auto_ptr`. It is deprecated in C++11 and removed from C++17. The C++11 language standard introduces `std::unique_ptr` as a safer replacement for `std::auto_ptr`. Use `std::unique_ptr` instead of `std::auto_ptr`.

### Polyspace Implementation

Polyspace flags all instances of `std::auto_ptr` in your code, other than those in C style arrays.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using `std::auto_ptr`

This code shows how Polyspace flags `std::auto_ptr` in your code.

```
#include <cstdint>
#include <memory>
#include <vector>
#define AUTOPTROF(_TYPE) std::auto_ptr<_TYPE>
AUTOPTROF(int) v_int;                          // Noncompliant
typedef struct {
    std::auto_ptr<bool> vb;                    // Noncompliant
} T;
T vec;
typedef std::auto_ptr<int> my_int_auto_ptr;     // Noncompliant
void Fn() noexcept
{

    std::auto_ptr<std::int32_t> ptr1(new std::int32_t(10)); // Noncompliant
    std::unique_ptr<std::int32_t> ptr2 =
    std::make_unique<std::int32_t>(10);         // Compliant
    std::vector<std::auto_ptr<std::int32_t>> v;  // Noncompliant
}

int main(){
    //..
}
```

Polyspace flags the `std::auto_ptr` objects. Use `std::unique_ptr` instead of `std::auto_ptr`.

## Check Information

**Group:** Language support library
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-1-4

A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type

## Description

### Rule Definition

*A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type.*

### Rationale

You must deallocate pointers to array elements by using `delete[]` instead of `delete`.

A pointer to an array element being passed to a smart pointer of single object type results in undefined behavior. Consider this code:

```
typedef A cArr[10];
std::unique_ptr<A> smartPtr1{new cArr};  //Noncompliant
```

`cArr` and its elements require `delete[]`. However, `smartPtr1` attempts to deallocate by using `delete`, resulting in undefined behavior.

Consider the following alternatives:

- Avoid using smart pointers to a pointer to an item in an array of objects. Instead use:
  - `std::array`
  - `std::vector`
  - `std::shared_ptr<std::vector<T>>`
- `std::unique_ptr<T[]>` and the corresponding overloads for `std::make_unique` .

  As of C++17, you can use `std::shared_ptr<T[]>`. The corresponding overloads for `std::make_shared` are not introduced until C++20.

Creating a custom deleter capable of handling an array of objects for the smart pointer of a single object type is considered noncompliant with this rule. This alternative can be error-prone, might no longer be supported in C++17, and is superseded by alternatives such as `std::unique_ptr<T[]>`.

### Polyspace Implementation

Polyspace raises this defect when you pass a pointer pointing to an element in an array of objects to a smart pointer of a single object. Polyspace also raises this defect if you pass a C-style array to a smart pointer of a single object.

Polyspace raises this defect when these conditions are met:

- You create a smart pointer by using `std::unique_ptr<T>` or `std::shared_ptr<T>`.
- You create an array of objects by using a C-style array or you create an array by using `std::make_unique` or `std::make_shared`.

- You use a function member of the smart pointer such as `release()` or `get()` to obtain the pointer to pass to the smart pointer.

You can pass the pointer to a smart pointer in several ways, including using a copy constructor, move constructor, or the `reset()` member function of the smart pointer.

When using a copy or move constructor, Polyspace flags the checker on the declared object name. In the case of a `reset()` member function, Polyspace flags the checker on the `reset()` member function.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Passing Pointer of Element of an Array to Smart Pointer of Single Object

```
#include <cstdint>
#include <vector>
#include <memory>

size_t x = 10;
class A
{
};

void example()
{
    std::unique_ptr<A[]> uPoint = std::make_unique<A[]>(x);
    std::shared_ptr<A> smartPtr1{ uPoint.get() };            //Noncompliant

    std::unique_ptr<A> smartPtr2;
    smartPtr2.reset(uPoint.release());                        //Noncompliant

    std::shared_ptr<std::vector<A>> smartPtr3;                //Compliant
}
```

There are two noncompliant smart pointers in the preceding example, `smartPtr1` and `smartPtr2`.

Both examples are noncompliant as, in each, a pointer to an array element is passed to a smart pointer that manages a single object, resulting in undefined behavior.

### C-Style Array Compliant By Using Compliant Standard Library Template

```
#include <cstdint>
#include <iostream>
#include <memory>

class A {
};

void Ex2()
{
    typedef A cArr[10];
    std::unique_ptr<A> smartPtr1{ new cArr };      //Noncompliant
    std::shared_ptr<A> smartPtr2{ new cArr };      //Noncompliant

    cArr a;
    std::unique_ptr<A> smartPtr3{ a };             //Noncompliant
    std::unique_ptr<A[]> smartPtr4{ a };           //Compliant
}
```

Because the C-style array is passed to the constructor of a smart pointer of a single object, Polyspace flags it as noncompliant. It does not matter if you create the smart pointer by using `std::unique_ptr` or `std::shared_ptr`. Both options are noncompliant.

Use of a specialized standard library template such as `std::unique_ptr<T[]>` as used in the example is compliant as long as you are using a compatible code version.

## Check Information

**Group:** Language support library
**Category:** Required, Automated

# Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-1-6

All std::hash specializations for user-defined types shall have a noexcept function call operator

## Description

### Rule Definition

*All std::hash specializations for user-defined types shall have a noexcept function call operator.*

### Rationale

`std::hash` specializations provided by the standard library have a guarantee of no exceptions. If you manually create a `std::hash` specialization, emulate this guarantee for your specialization. Define all specializations of `std::hash` for your custom data types as `noexcept`.

Otherwise, standard library containers that use your specialization of `std::hash` indirectly might throw uncaught exceptions. The exceptions are not caught because the standard library containers do not provide a way to use `try-catch` blocks for exceptions from `std::hash`.

### Polyspace Implementation

The checker flags specializations of the `std::hash` template with user defined types that do not have a `noexcept` specifier.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language support library
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A18-5-1

Functions malloc, calloc, realloc and free shall not be used

## Description

### Rule Definition

*Functions malloc, calloc, realloc and free shall not be used.*

### Rationale

C-style memory allocation and deallocation using `malloc`, `calloc`, `realloc`, or `free` is not type safe and does not invoke class's constructors/destructor to create/delete objects.

For instance, `malloc` allocates memory to an object and returns a pointer to the allocated memory of type `void*`. A program can then implicitly cast the returned pointer to a different type that might not match the intended type of the object.

The use of these allocation and deallocation functions can result in undefined behavior if:

- You use `free` to deallocate memory allocated with operator `new`.
- You use operator `delete` to deallocate memory allocated with `malloc`, `calloc`, or `realloc`.

The rule is not violated when you perform dynamic memory allocation or deallocation using overloaded `new` and `delete` operators, or custom implementations of `malloc` and `free`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Non-Compliant Use of `malloc`

```
#include <cstdint>
#include <cstdlib>

void func()
{

    std::int32_t* p1 = static_cast<std::int32_t*>(malloc(sizeof(std::int32_t))); // Non-compliant
    *p1 = 0;

    free(p1); // Non-compliant

    std::int32_t* p2 = new std::int32_t(0); // Compliant

    delete p2; // Compliant
}
```

In this example, the allocation of memory for pointer `p1` using `malloc` and the memory deallocation using `free` are non-compliant. These operations are not type safe. Instead, use operators `new` and `delete` to allocate and deallocate memory.

## Check Information

**Group:** 18 Language Support Library
**Category:** Required, Automated

# Version History

**Introduced in R2019b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-5-2

Non-placement new or delete expressions shall not be used

## Description

### Rule Definition

*Non-placement new or delete expressions shall not be used.*

### Rationale

Explicit use of nonplacement `new` or `delete` operators might result in memory leaks caused by unexpected exceptions or returns. Consider this code where memory is allocated for a pointer by explicitly calling `new` and deallocated by explicitly calling `delete`.

```
std::int32_t ThrowError(){
    std::int32_t errorCode;
    std::int31_t* ptr = new std::int32_t{0};
    //...
    if(errorCode!=0){
        throw std::runtime_error{"Error"};
    }
    //...
    if (errorCode != -1) {
        return 1;
    }
    delete ptr;
    return errorCode;
}
```

This code can lead to unexpected memory leak in certain conditions.

- If the first `if()` statement is `true`, then the function produces an exception and exits without deleting the pointer.
- If the second `if()` statement is `true`, then the function returns `1` and exits without deleting the pointer.

To avoid an unpredictable memory leak, do not use nonplacement `new` and `delete` operators. Instead, encapsulate dynamically allocated resources in objects. Acquire the resources in object constructors and release the resources in object destructors. This design pattern is called "Resource Acquisition Is Initialization" or RAII. Following the RAII pattern prevents a memory leak even when there are unexpected exceptions and returns.

Alternatively, use manager objects that manage the lifetime of dynamically allocated resources. Examples of manager objects in the standard library include:

- `std::unique_ptr` along with `std::make_unique`
- `std::shared_ptr` along with `std::make_shared`
- `std::string`
- `std::vector`

This rule does not apply to a `new` operator or a `delete` operator in user-defined RAII classes and managers.

### Polyspace Implementation

AUTOSAR C++14 permits explicit resource allocation by calling the `new` operator in two cases, when the allocated resource is immediately passed to:

- A manager object
- A RAII class that does not have a safe alternative to the `new` operator.

Polyspace flags all explicit uses of the `new` operator and the `delete` operator. If you have a process where a `new` operator can be permissible and there is no safer alternative, justify the issue by using comments in your result or code. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Explicitly Calling `new` Operator and `delete` Operator

This code shows how Polyspace flags `new` or `delete` operators.

```
#include <cstdint>
#include <memory>
#include <vector>
#include <cstddef>

using namespace std;

int32_t Fn1()
{
    int32_t errorCode{0};
    int32_t* ptr =
    new int32_t{0}; //Noncompliant
    // ...
    if (errorCode != 0) {
        throw  runtime_error{"Error"}; // Possible Memory Leak
    }
    // ...
    if (errorCode != 0) {
        return 1; //Possible Memory Leak
    }
    // ...
    delete ptr; //Noncompliant
    return errorCode; // Possible Memory Leak
}

int32_t Fn2()
{
    int32_t errorCode{0};
    // Alternative to 'new'
    unique_ptr<int32_t> ptr1 = make_unique< int32_t>(0);
    unique_ptr<int32_t> ptr2(new  int32_t{0});   // Noncompliant
    shared_ptr<int32_t> ptr3 =
    make_shared<int32_t>(0);   //Compliant
    vector<int32_t> array;   // Compliant

    if (errorCode != 0) {
        throw  runtime_error{"Error"};   // No memory leaks
    }
    // ...
    if (errorCode != 0) {
        return 1; // No memory leaks
    }
    // ...
    return errorCode; // No memory leaks
}


class X
{
public:
    static void* operator new( size_t s)
    {
        return ::operator new(s);   // Noncompliant
    }

    static void* operator new[]( size_t s)
    {
        return ::operator new(s);   // Noncompliant
    }

    static void operator delete(void* ptr,  size_t s)
    {
        ::operator delete(ptr);   // Noncompliant
    }

    static void operator delete[](void* ptr,  size_t s)
    {
        ::operator delete(ptr);   // Noncompliant
    }
};

main(){
    X* x1    = new X;   // Noncompliant
    X* x2    = new X[2];   // Noncompliant
}
```

In `Fn1()`, the operators `new` and `delete` are explicitly called for resource management. Consequently, an unexpected exception or return can lead to a memory leak. Polyspace flags the `new` and `delete` operators. In `Fn2()`, manager objects are used for memory management. Even in cases of unexpected exceptions and returns, there are no memory leaks in `Fn2()`.

The class `X` contains custom overloads for `new` and `delete` operators. Polyspace flags all instances of `new` and `delete` operators in the definitions of the custom overloads. In main(), Polyspace also flags the overloaded new and delete operators.

## Check Information

**Group:** Language support library
**Category:** Required, Partially automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-5-3

The form of delete operator shall match the form of new operator used to allocate the memory

## Description

### Rule Definition

*The form of delete operator shall match the form of new operator used to allocate the memory.*.

### Rationale

- The `delete` operator releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.
- If you use the single-object notation for `delete` on a pointer that is previously allocated with the array notation for `new`, the behavior is undefined.

The issue can also highlight other coding errors. For instance, you perhaps wanted to use the `delete` operator or a previous `new` operator on a different pointer.

### Polyspace Implementation

The checker flags a defect when:

- You release a block of memory with the `delete` operator but the memory was previously not allocated with the `new` operator.
- You release a block of memory with the `delete` operator using the single-object notation but the memory was previously allocated as an array with the `new` operator.

This defect applies only to C++ source files.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Deleting Static Memory

```
void assign_ones(void)
{
    int ptr[10];

    for(int i=0;i<10;i++)
        *(ptr+i)=1;

    delete[] ptr;
}
```

The pointer `ptr` is released using the `delete` operator. However, `ptr` points to a memory location that was not dynamically allocated.

#### Correction: Remove Pointer Deallocation

If the number of elements of the array ptr is known at compile time, one possible correction is to remove the deallocation of the pointer ptr.

```
void assign_ones(void)
{
    int ptr[10];

    for(int i=0;i<10;i++)
        *(ptr+i)=1;
}
```

#### Correction — Add Pointer Allocation

If the number of array elements is not known at compile time, one possible correction is to dynamically allocate memory to the array ptr using the new operator.

```
void assign_ones(int num)
{
    int *ptr = new int[num];

    for(int i=0; i < num; i++)
        *(ptr+i) = 1;

    delete[] ptr;
   }
```

### Mismatched new and delete

```
int main (void)
{
    int *p_scale = new int[5];

    //more code using scal

    delete p_scale;
}
```

In this example, p_scale is initialized to an array of size 5 using new int[5]. However, p_scale is deleted with delete instead of delete[]. The new-delete pair does not match. Do not use delete without the brackets when deleting arrays.

#### Correction — Match delete to new

One possible correction is to add brackets so the delete matches the new [] declaration.

```
int main (void)
{
    int *p_scale = new int[5];

    //more code using p_scale

    delete[] p_scale;
}
```

#### Correction — Match new to delete

Another possible correction is to change the declaration of p_scale. If you meant to initialize p_scale as 5 itself instead of an array of size 5, you must use different syntax. For this correction, change the square brackets in the initialization to parentheses. Leave the delete statement as it is.

```
int main (void)
{
    int *p_scale = new int(5);

    //more code using p_scale

    delete p_scale;
}
```

## Check Information

**Group:** Language Support Library
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

Check AUTOSAR C++ 14 (-autosar-cpp14)

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-5-4

If a project has sized or unsized version of operator 'delete' globally defined, then both sized and unsized versions shall be defined

## Description

### Rule Definition

*If a project has sized or unsized version of operator 'delete' globally defined, then both sized and unsized versions shall be defined.*

### Rationale

The C++14 Standard defines a sized version of `operator delete`. For instance, for an unsized `operator delete` with this signature:

```
void operator delete (void* ptr);
```

The sized version has an additional size argument:

```
void operator delete (void* ptr, std::size_t size);
```

See the C++ reference page for `operator delete`.

The Standard states that if both versions of `operator delete` exist, the sized version must be called because it provides a more efficient way to deallocate memory. However, in some cases, for instance to delete incomplete types, the unsized version is used.

If you overload the unsized version of `operator delete`, you must also overload the sized version. You typically overload `operator delete` to perform some bookkeeping in addition to deallocating memory on the free store. If you overload the unsized version but not the sized one or the other way around, any bookkeeping you perform in one version will be omitted from the other version. This omission can lead to unexpected results.

### Polyspace Implementation

The checker flags situations where an unsized version of `operator delete` exists but the corresponding sized version is not defined, or vice versa.

The checker is enabled only if you specify a C++ version of C++14 or later. See `C++ standard version (-cpp-version)`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Missing Sized Overload of `operator delete[]`

```
#include <new>
#include <cstdlib>
```

```
int global_store;

void update_bookkeeping(void *allocated_ptr, bool alloc) {
   if(alloc)
      global_store++;
   else
      global_store--;
}

void operator delete(void *ptr);
void operator delete(void* ptr) {
    update_bookkeeping(ptr, false);
    free(ptr);
}

void operator delete(void *ptr, std::size_t size);
void operator delete(void* ptr, std::size_t size) {
    //Compliant, both sized and unsized version defined
    update_bookkeeping(ptr, false);
    free(ptr);
}

void operator delete[](void *ptr);
void operator delete[](void* ptr) { //Noncompliant, only unsized version defined
    update_bookkeeping(ptr, false);
    free(ptr);
}
```

In this example, both the unsized and sized version of `operator delete` are overloaded and complies with the rule. However, only the unsized version of `operator delete[]` is overloaded, which violates the rule..

## Check Information

**Group:** Language Support Library
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Invalid free of pointer` | `Invalid deletion of pointer` | `Memory leak` | `Mismatched alloc/dealloc functions on Windows` | `Missing overload of allocation or deallocation function` | `Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-5-5

Memory management functions shall ensure the following: (a) deterministic behavior resulting with the existence of worst-case execution time, (b) avoiding memory fragmentation, (c) avoid running out of memory, (d) avoiding mismatched allocations or deallocations, (e) no dependence on non-deterministic calls to kernel

## Description

### Rule Definition

*Memory management functions shall ensure the following: (a) deterministic behavior resulting with the existence of worst-case execution time, (b) avoiding memory fragmentation, (c) avoid running out of memory, (d) avoiding mismatched allocations or deallocations, (e) no dependence on non-deterministic calls to kernel.*

### Rationale

When you implement custom memory management functions, make sure that your implementation addresses these common memory management errors that can affect the stability and correctness of your application:

- Non-deterministic worst-case execution time (WCET) of allocation and deallocation operations.

  To provide a deterministic WCET, make sure that the function can be executed without context switching or system calls. A predictable WCET is essential in determining an appropriate scheduling scheme that meets the timing constraints in safety-critical embedded systems.
- Mismatched allocation and deallocation functions.

  Deallocating memory with a function that does not match the allocation function can cause memory corruption or undefined behavior.
- Invalid memory access.

  If you try to access memory that is logically or physically invalid, the operation results in undefined behavior or a segmentation fault.
- Out-of-memory errors.

  To avoid running out of memory, your executable should allocate all the memory needed by the program at startup.
- Memory fragmentation.

  Fragmentation occurs when memory is allocated over non-contiguous blocks. If the unallocated blocks are not large enough to accommodate future allocation requests, the remaining free memory might not be usable and your system might crash.

In addition to custom implementations for operators `new` and `delete`, you should provide custom implementations for low-level allocation and deallocation functions (`malloc/free`). Even if you do not use these low-level functions in your source code, they can occur in linked libraries in your project.

A custom implementation of `std::new_handler` must perform one of these operations:

- Make more memory available for allocations and return.
- Terminate the program without returning to the callee.
- Throw an exception of type `std::bad_alloc` or derived from `std::bad_alloc`.

### Polyspace Implementation

Polyspace checks for these memory management issues that might result in non-deterministic behavior:

- Use of C library function (`malloc/calloc/realloc/free`) to allocate or deallocate memory in local variable initializations.

  Polyspace does not flag the use of these functions when allocating or deallocating global variables.
- Use of non-placement `new` and `delete` operators.
- Use of function `dlsym()`. This function might call low-level allocation or deallocation functions such as `malloc` or `calloc`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Non-Deterministic Memory Management Functions

```
#define _GNU_SOURCE
#include <malloc.h>
#include <dlfcn.h>
#include <iostream>

class Point
{
public:
    Point(int x1 = 0, int y1 = 0): x(x1), y(y1)
    {
    }
    ~Point();
private:
    int x, y;
};


void func1()
{
    unsigned char buffer[sizeof(int) * 2];
    Point* p1 = new Point(0, 0); // Non-compliant
    Point* p2 = new (buffer) Point(1, 1); // Compliant

    int* p3 = (int*)malloc(sizeof(int)); // Non-compliant

    //Use pointers

    delete p1; // Non-compliant
    p2->~Point();
    free(p3); // Non-compliant
}

void* customAlloc(size_t size)
{
    void* (*myAlloc)(size_t) =
            (void* (*)(size_t))dlsym(RTLD_NEXT, "malloc"); // Non-compliant

    return myAlloc(size);

}
```

In this example, Polyspace flags these dynamic memory management operations:

- The allocation and deallocation of pointer `p1` with non-placement operators `new` and `delete`.
- The allocation and deallocation of local pointer `p3` with low-level functions `malloc` and `free`. Polyspace does not flag the use of these functions when allocating or deallocating global variables.
- The use of function `dlsym()` because the function calls `malloc`.

Polyspace does not flag the initialization of pointer `p2` because it uses a placement operator `new` which creates the pointer in a pre-allocated buffer.

## Check Information

**Group:** Language support library
**Category:** Required, Partially automated

# Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `Invalid deletion of pointer` | `Mismatched alloc/dealloc functions on Windows` | `Invalid free of pointer`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A18-5-7

If non-real-time implementation of dynamic memory management functions is used in the project, then memory shall only be allocated and deallocated during non-real-time program phases

## Description

### Rule Definition

*If non-real-time implementation of dynamic memory management functions is used in the project, then memory shall only be allocated and deallocated during non-real-time program phases.*

### Rationale

A real-time function is one with a known worst case execution time. That is, the execution time of real-time functions cannot exceed a specific and known value.

Inside a real-time function, you might be using functions that manage dynamic memory, such as `new` or `delete`. The execution time of these functions depends on how much memory the functions manage. Because their worst case execution time is not deterministic, using these functions in the real time phase of the application might result in unexpected behaviors, memory leaks, and memory fragmentation. Dynamic memory management in real time requires implementing deterministic implementations of these functions that have a known worst case execution time.

Avoid using non-real-time dynamic memory management functions in the real time phase of your application. Perform non-real-time memory operations in the non-real-time phase such as the initialization or the non-real-time state transitions.

### Polyspace Implementation

To check for violations of this rule, specify your real-time functions by using the analysis option `-code-behavior-specifications`. In the code behavior specification XML file, specify a function as a real-time function by using the behavior `REAL_TIME_FUNC`. Polyspace flags a specified real-time function if :

- The function allocates or deallocates dynamic memory by using a non-real-time implementation.
- The function calls a function that uses non-real-time dynamic memory management.

Polyspace assumes that these functions from the standard library use non-real-time implementation of dynamic memory management:

- The operators `new` and `delete`.
- `std::make_unique()`
- `std::vector::vector()`
- `std::vector::reserve()`
- `std::basic_string::basic_string()`

You might use other functions in your code that use non-real-time implementation of dynamic memory management. Specify these functions as non-real-time dynamic memory management function by using the behavior `MANAGES_MEMORY`.

To use this rule, specify at least one entry that has the behavior `REAL_TIME_FUNC`. If you use this checker without specifying the code behavior, Polyspace produces a warning.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Non-Real-Time Memory Operations in Real-Time Functions

```
#include <cstdlib>
#include <new>
#include<vector>
extern bool pollSensor();
extern int getSize();
extern int getSensorData();
int* allocateIntArray(int size){
    return new int[size];  //Noncompliant
}
void deallocateIntArray(int* data){
    delete data;  //Noncompliant
}
void AppMain(){
    int* Data = allocateIntArray(getSize());
    int index = 0;
    while(1){
        if(pollSensor()){
            Data[index] = getSensorData(); ;
            //...

        }
        deallocateIntArray(Data);
    }
}
void AppMainAlt(){
    std::vector<int> Data;  //Noncompliant
    while(1){
        if(pollSensor()){
            Data.push_back(getSensorData());
        }
    }
}
```

To run this example, specify the functions `AppMain` and `AppMainAlt` as real-time functions. For instance, save this the code behavior specifications XML code as *`code_behavior.xml`*:

```
<specifications>
    <functions>
        <function name="AppMain">
            <behavior name="REAL_TIME_FUNC"/>
        </function>
        <function name="AppMainAlt">
            <behavior name="REAL_TIME_FUNC"/>
        </function>
    </functions>
</specifications>
```

After specifying the code behavior, add this option when running the analysis:

In this example, Polyspace flags non-real-time memory management in real-time functions.

- The duration for the operation `new` in `allocateIntArray()` depends on the argument `size` and cannot be predetermined. Using this function in real time might result in a memory leak or a segmentation fault. Because `allocateIntArray()` is called from the real-time function `AppMain`, Polyspace raises a violation.
- The non-real-time deallocation operation in `deallocateIntArray()` might result in a memory leak or a segmentation fault if the function is used in real time. Because `deallocateIntArray()` is called from the real-time function `AppMain`, Polyspace flags it.
- Polyspace flags the non-real-time memory allocation operation `allocateIntArray` in the real-time function `AppMain`.
- Polyspace flags the vector declaration, which is a non-real-time memory allocation operation, in the real-time function `AppMainAlt`.

### Perform Dynamic Memory Allocation In Non-Real-Time Phase

```
#include <cstdlib>
#include <new>
#include<vector>
extern bool pollSensor();
extern int getSize();
extern int getSensorData();

void AppMain(std::vector<int>& Data){
    while(1){
        if(pollSensor()){
            Data.push_back(getSensorData());
        }
    }
}
int main(){
    std::vector<int> Data;
    Data.reserve(1024);
    AppMain(Data);
    return 0;
}
```

This example shows a best practice when using non-real-time memory management in a program that uses real-time functions. The memory allocation operation is performed in the function `main()`, which is not executed at real time. The real-time function is `AppMain()`, and this function does not perform any dynamic memory management. Separating memory management from real time functions might reduce the possibility of memory leaks, segmentation faults, and other unexpected behaviors.

## Check Information

**Group:** Language support library
**Category:** Required, Non-automated

## Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `-code-behavior-specifications`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-5-8

Objects that do not outlive a function shall have automatic storage duration

## Description

### Rule Definition

*Objects that do not outlive a function shall have automatic storage duration.*

### Rationale

A dynamically allocated object results in additional allocation and deallocation costs and makes your program vulnerable to memory leaks if, for instance, the program returns due to an exception throw before the deallocation operation.

Instead, use an object with automatic storage duration, which has a lifetime that is bound to the enclosing scope of that object. The object is automatically destroyed when that scope exits.

The rule allows an exception for local objects that are dynamically allocated to optimize stack memory usage because the objects use a large amount of memory and might otherwise cause a stack overflow.

### Polyspace Implementation

Polyspace flags objects that are created in a function scope and that do not have automatic storage duration when any of the following is true:

- The object is a smart pointer (`std::shared_ptr` or `std::unique_ptr`) that is never copied, moved, reassigned, reset, or passed to a callee.

  The object is not flagged if it is a non-array and, at compilation time, its size is greater than 4 KB or its size is unknown.
- The object is dynamically allocated by using operators `new` or `new[]` and then deallocated through all possible paths within the function.

  The object is not flagged if it is a non-array and, at compilation time, its size is greater than 4 KB or its size is unknown.
- The object is a wrapper class that contains at least one data member with a fixed size larger than 16 KB.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unnecessary Use of a Smart Pointer

```
#include <iostream>
#include <cstdint>
```

```
#include <array>
#include <utility>

constexpr std::size_t size4KB = 4 * 1024;

class MyLargeBoard
{
  public:
    constexpr static size_t size16KB = 16 * 1024;
    MyLargeBoard() {}
    MyLargeBoard(int s);
  private:
    std::array<uint8_t, size16KB> cells;

};

void Reset_ptr(std::shared_ptr<std::pair<int32_t, int32_t>>& ptr)
{
    ptr.reset();
}

void Func(int32_t x_coord, int32_t y_coord,
          std::pair<int32_t, int32_t>** param_ptr)
{
    std::shared_ptr<std::pair<int32_t, int32_t>>reused_ptr(*param_ptr);

    auto toResetPtr = //Compliant, smart pointer is reset
        std::make_shared<std::pair<int32_t, int32_t>>(x_coord, y_coord);
    auto unused_ptr = //Non-compliant
        std::shared_ptr<std::pair<int32_t, int32_t>>(reused_ptr);

    if (toResetPtr->first || toResetPtr->second) {
        Reset_ptr(toResetPtr);
    }

}

void SmartPtrLargeMem()
{
    //Large non-array object
    std::shared_ptr<MyLargeBoard>
        big_non_array(new MyLargeBoard); // Compliant
    //Large array of char
    std::unique_ptr<char []>
        big_array {new char[size4KB]{'1', '2', '3', '4'}}; //Non-compliant

}
```

In this example, Polyspace flags these smart pointers as noncompliant:

- Shared smart pointer `unused_ptr` because it is declared locally in `Func` and it is never copied, moved, reassigned, reset, or passed to a callee.
- `big_array` which manages the dynamically allocate array of `char` in `SmartPtrLargeMem`. In this context, the use of a standard C++ container such as `std::array` instead of `std::unique_ptr` is less memory intensive.

Polyspace does not flag:

- `reused_ptr` because it is used to initialize `unused_ptr` and `toResetPtr` because it is reset.
- `big_non_array` because it is a non-array object with one data member of size smaller than 16 KB. The rule allows an exception for such objects because dynamic allocation can help optimize stack memory usage, for instance on an embedded device with limited memory storage.

### Unnecessary Dynamic Memory Allocation

```
#include <iostream>
#include <cstdint>
#include <array>
#include <utility>

constexpr std::size_t size4KB = 4 * 1024;

class MyLargeBoard
```

```
{
public:
    constexpr static size_t size16KB = 16 * 1024;
    MyLargeBoard() {}
    MyLargeBoard(int s);
private:
    std::array<uint8_t, size16KB> cells;

};


class Mytype
{
public:
    Mytype(int s = 0) : a{s} {}
private:
    int a;
};

void ReadInput(Mytype* input);
bool IsInValidRange(Mytype* input);

void* func(Mytype** output)
{
    auto input1 = new Mytype(); //Non-compliant
    ReadInput(input1);

    auto input2 = new Mytype(); //Compliant
    if (IsInValidRange(input2)) {
        delete input2;
    } else {
        *output = input2;
    }


    Mytype input3; //Compliant
    ReadInput(&input3);

    delete input1;

    return nullptr;

}

void DynamicAllocLargeMem()
{
    //Large non-array object
    MyLargeBoard* big_non_array {new MyLargeBoard}; //Compliant
    //Large array of char
    char* big_array { new char[size4KB]{'1', '2', '3', '4'}}; //Non-compliant
    // ....

    delete big_non_array;
    delete[] big_array;

}
```

In this example, noncompliant local variable `input1` is dynamically allocated with operator `new` and then deallocated through all possible paths inside `func`. The unnecessary allocation and deallocation operations can be avoided by declaring a variable with automatic storage duration, such as `input3`, which is automatically deleted when `func` returns .

Polyspace also flags dynamically allocated array `big_array`. In this context, the use of a standard C ++ container such as `std::array` is less memory intensive.

Dynamically allocated variable `input2` is compliant because it is not deallocated though all possible paths inside `func`. The variable is escaped through `output` in the `else` branch.

Similarly, Polyspace does not flag `big_non_array` because it is a non-array object with one data member of size smaller than 16 KB. The rule allows an exception for such objects because dynamic allocation can help optimize stack memory usage, for instance on an embedded device with limited memory storage.

### Use of Wrapper Class With Large Size Data Members

```
#include <iostream>
#include <cstdint>
#include <array>
#include <vector>

class MyTable
{

public:
    constexpr static size_t size64KB = 65535;
    using arrayType = std::array<uint8_t, size64KB>;
    MyTable() :   tableSize{0} {}
    MyTable(const std::string& dbPath, uint32_t inputSize) : tableSize{inputSize}
    {

        // ...
    }
    uint8_t AverageCellVal() const noexcept;
private:
    arrayType table;
    uint32_t tableSize;
};

void AvgCellVal(const std::string& dbPath, uint32_t inputSize)
{
    std::vector<uint8_t> table1(inputSize);  // Compliant
    MyTable table2(dbPath, inputSize); // Non-compliant
    uint8_t averageCellV = table2.AverageCellVal();
    std::cout << "Average cell value in " << dbPath << ": " << averageCellV << '\n';
}

class DerivedMyTable : public MyTable
{
public:
    constexpr static size_t pathMaxSize = 2 * 1024; // 2 Kb
    DerivedMyTable() : MyTable() {}
    DerivedMyTable(const std::string& dbPath, uint32_t inputSize) : MyTable(dbPath, inputSize)
    {
        std::strncpy(__dbPath, dbPath.data(), pathMaxSize - 1);
        __dbPath[pathMaxSize - 1] = '\0';
    }
private:
    char __dbPath[pathMaxSize];  // OK
};


void func(const std::string& dbPath, uint32_t inputSize)
{
    DerivedMyTable table2_derived(dbPath, inputSize);  // Non-compliant

}
```

In this example, base class `MyTable` contains a data member `table` of type `arrayType` which corresponds to an array of size 64 KB. Polyspace flags the declaration of variable `table2` because it consumes a large amount of memory through `MyTable` and it is only used within `AvgCellVal()`.

Instead, you can use a less memory intensive C++ container such as `std::vector` to declare an automatic storage duration object such as `table1`.

Polyspace also flags `table2_derived`. Even if `DerivedMyTable` does not contain a data member with a large size in memory, it is derived from a base class that wraps an object that consumes a large size of memory.

## Check Information

**Group:** Language support library
**Category:** Required, Partially automated

## Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-5-9

Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard

## Description

### Rule Definition

*Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard.*

### Rationale

The C++ Standard (*[new.delete]*) specifies certain required behaviors for the dynamic allocation and deallocation functions. If you implement a global replacement allocation or deallocation function that does not meet these semantic requirements, other functions that rely on the required behaviors might behave in an undefined manner.

For instance, `void* operator new ( std::size_t count )` is expected to throw a `bad_alloc` exception if it fails to allocate the requested amount of memory. If you implement a replacement allocation function that returns `nullptr` instead of throwing, a function that expect the memory allocation to throw on failure might try to dereference a null pointer instead.

### Polyspace Implementation

Polyspace flags these replacement implementations of dynamic allocation and deallocation functions.

- Replacement `operator new` that returns `nullptr` when the expected behavior is to throw a `bad_alloc` exception on failure.
- Replacement `operator new` or `operator delete` that throw directly or indirectly on failure when the expected behavior is to not throw. Polyspace also highlights the location of the throw in your code.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### `operator new` Returns `nullptr` on Failure

```
#include<new>

extern void* custom_alloc(std::size_t);

void* operator new (std::size_t count) //Non-compliant
{
    return custom_alloc(count);
}

void func()
```

```
{
    int* ptr1;
    try {
        ptr1 = new int;
    } catch (const std::bad_alloc&) {
        //handle exception
    }

    //Use ptr1

}
```

In this example, the custom allocation function `custom_alloc`, which is defined elsewhere, might return `nullptr` on failure. Function `func`, which expects a `bad_alloc` exception if the memory allocation fails, might dereference a null pointer because `operator new` does not throw.

## Check Information

**Group:** Language support library
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-5-10

Placement new shall be used only with properly aligned pointers to sufficient storage capacity

## Description

### Rule Definition

*Placement new shall be used only with properly aligned pointers to sufficient storage capacity.*

### Rationale

The `new` operator allocates the required amount of memory for storing an object on the heap and constructs a new object in the allocated memory in a single operation. If you want to separate the allocation and the construction and place an object in preallocated memory on either the stack or the heap, you use placement `new`. Placement `new` has advantages over `new` in certain situations, for example, when you need to place the object at a known memory location.

The `new` operator automatically allocates the correct amount of aligned memory that the object requires. But when using placement `new`, you must manually make sure that the pointer you pass has sufficient allocated storage capacity and is properly aligned. Violating these constraints results in the construction of an object at a misaligned location or memory initialization outside of allocated bounds, which might lead to unexpected or implementation-dependent behavior.

### Polyspace Implementation

Suppose that a pointer `ptr` is preallocated `m` bytes of memory on the stack and has alignment `n`. For instance, if `ptr` is an array:

```
uint8_t ptr[5];
```

the allocated storage is `sizeof(uint8_t) * 5` and the alignment is `alignof(uint8_t)`. If you allocate more than `m` bytes to this pointer in a placement `new` expression or if the alignment required for the allocation is greater than `n`, the checker raises a violation. When determining the pointer alignment, the checker takes into account explicit alignments such as with `std::align`.

The checker does not consider pointers that are preallocated memory on the heap since the available storage depends on the memory availability, which is known only at run time.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Placement `new` Used with Insufficient Storage Capacity and Misaligned Pointers

```
#include <new>
#include<memory>
#include <cstdint>

void Foo()
```

```
{
  uint8_t c;
  uint64_t* ptr =
      new    // Non-compliant (insufficient storage, misaligned)
      (&c) uint64_t;
}

void Bar()
{
  uint8_t buf[sizeof(uint64_t)];
  uint64_t* ptr =
      new            // Non-compliant (sufficient storage, misaligned)
      (buf) uint64_t;
}

void Baz()
{
  void* buf;
  std::size_t sp = 64;
  std::align(alignof(uint64_t), sizeof(uint64_t), buf, sp);
  uint64_t* ptr =
      new            // Compliant (sufficient storage, aligned)
      (buf) uint64_t;
}
```

In the function `Foo`, the `&c` points to an `uint8_t` value and has one byte memory in stack with one-byte alignment. The pointer is passed to placement `new`, which constructs an instance of `uint64_t` that requires 8 bytes of memory and a 4-byte alignment. This usage violates the rule.

In the function `Bar`, the pointer `buf` is properly allocated and has sufficient storage capacity. But, because it points to the `uint8_t` data type, it has one-byte alignment. This usage still violates the rule.

The function `Baz` calls the `std::align` function to create a pointer with correct storage capacity (8 byte) and alignment (4-byte) for `uint64_t`. This usage complies with the rule.

## Check Information

**Group:** Language support library
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-5-11

"operator new" and "operator delete" shall be defined together

## Description

### Rule Definition

*"operator new" and "operator delete" shall be defined together.*

### Rationale

You typically overload `operator new` to perform some bookkeeping in addition to allocating memory on the free store. Unless you overload the corresponding `operator delete`, it is likely that you omitted some corresponding bookkeeping when deallocating the memory.

The defect can also indicate a coding error. For instance, you overloaded the placement form of `operator new[]`:

```
void *operator new[](std::size_t count, void *ptr);
```

but the non-placement form of `operator delete[]`:

```
void operator delete[](void *ptr);
```

instead of the placement form:

```
void operator delete[](void *ptr, void *p );
```

When overloading `operator new`, make sure that you overload the corresponding `operator delete` in the same scope, and vice versa. To find the `operator delete` corresponding to an `operator new`, see the reference pages for `operator new` and `operator delete`.

### Polyspace Implementation

The rule checker raises a violation when you overload `operator new` but do not overload the corresponding `operator delete`, or vice versa.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Mismatch Between Overloaded `operator new` and `operator delete`

```
#include <new>
#include <cstdlib>

int global_store;

void update_bookkeeping(void *allocated_ptr, bool alloc) {
   if(alloc)
```

```
      global_store++;
   else
      global_store--;
}


void *operator new(std::size_t size, const std::nothrow_t& tag);
void *operator new(std::size_t size, const std::nothrow_t& tag) //Noncompliant
{
    void *ptr = (void*)malloc(size);
    if (ptr != nullptr)
        update_bookkeeping(ptr, true);
    return ptr;
}

void operator delete[](void *ptr, const std::nothrow_t& tag);
void operator delete[](void* ptr, const std::nothrow_t& tag) //Noncompliant
{
    update_bookkeeping(ptr, false);
    free(ptr);
}
```

In this example, the overloads of operators `operator new` and `operator delete[]` are noncompliant because there are no overloads of the corresponding `operator delete` and `operator new[]` operators.

The overload of `operator new` calls a function `update_bookkeeping` to change the value of a global variable `global_store`. If the default `operator delete` is called, this global variable is unaffected, which might defy developer's expectations.

### Check Information

**Group:** Language support library
**Category:** Required, Automated

## Version History

**Introduced in R2020b**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-9-1

The std::bind shall not be used

## Description

### Rule Definition

*The std::bind shall not be used.*

### Rationale

`std::bind` takes a callable object, such as a function object, and produces a forwarding call wrapper for this object. Calling the wrapper invokes the object with some of the object arguments bound to arguments you specify in the wrapper. For instance, in this code snippet, `foo` is called through `bar` with the first (second) argument of `bar` bound to the second (first) argument of `foo`.

```
int foo(int, int);
auto bar = std::bind(foo, _2, _1);
bar(10, 20); //call to foo(20, 10)
```

The use of `std::bind` results in a less readable function call. A developer that is unfamiliar with `foo` would need to see the declaration of `foo` to understand how to pass arguments to `bar`, and might confuse one function parameter with another. In addition, a compiler is less likely to inline a function that you create using `std::bind`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Non-Compliant Use of `std::bind`

```
#include <cstdint>polys
#include <functional>
class A
{
//...
};
void func(A const& a, double y) noexcept
{
//...
}
void func1() noexcept
{
    double arg2 = 0.0;
    auto bind_fn = std::bind(&func, std::placeholders::_1, arg2); // Non-compliant
    // ...
    A const a{};
    bind_fn(a);
}
void func2() noexcept
{
    auto lambda_fn = [](A const & a) -> void { // Compliant
        double arg2 = 0.0;
        func(a, arg2);
    }; // Compliant
    // ...
    A const a{};
    lambda_fn(a);
}
```

In this example, `func` is called through `bind_fn` with the only argument of `bind_fn` bound to the first argument of `func`. It might be unclear to a developer that `arg2` in the definition of `bind_fn` is the second argument of `func`. For a more readable code, use lambda expressions instead. The call to `func` with two arguments is clearer in the definition of `lambda_fn`.

### Check Information

**Group:** 18 Language Support Library
**Category:** Required, Automated

## Version History

**Introduced in R2019b**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**
"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-9-2

Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference

## Description

### Rule Definition

*Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference.*

### Rationale

You can pass an object efficiently to a function by casting the object to an rvalue and taking advantage of move semantics.

- If you are forwarding an rvalue reference to a function, use `std::move` to cast the object to an rvalue.
- If you are forwarding a forwarding reference (or universal reference) to a function, use `std::forward` to cast the object to an rvalue if and only if the object is bound to an rvalue. A forwarding reference might be bound to an rvalue or an lvalue. For the purposes of this rule, objects with type `auto &&` are considered as forwarding references.

Using `std::move` with forwarding references might result in an unexpected modification of an lvalue. Using `std::forward` with rvalue references is possible but it is error-prone and might increase the complexity of your code.

### Polyspace Implementation

- Polyspace flags the use of `std::move` to forward a forwarding reference to a function, including objects of type `auto &&`.
- Polyspace flags the use of `std::forward` to forward an rvalue reference to a function.
- Polyspace does not flag the use of `std::move` or `std::forward` if no forwarding to a function takes place. For instance, in this code snippet, no defect is raised on the use of `std::move` with forwarding reference `b2` and the use of `std::forward` with revalue reference `b1`.

```
template <typename T1, typename T2>
void func(T1& b1, T2&& b2)
{
    const T1& b10 = std::forward<B>(b1);
    const T2& b20 = std::forward<B>(b2);
    const T1& b11 = std::move(b1);
    const T2& b21 = std::move(b2);
}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Values Forwarded Incorrectly

```
#include <cstdint>
#include <string>
#include <utility>


class A
{
public:
    explicit A(std::string&& s)
        : str(std::move(s)) // Compliant
    {
    }

private:
    std::string str;
};

template <typename ...T>
void f1(T...t);



template <typename T1, typename T2>
void func(T1&& t1, T2& t2)
{
    f1(std::move(t1));            // Non-compliant
    f1(std::forward<T1>(t1));     // Compliant

    f1(std::forward<T2>(t2));     // Non-compliant
    f1(std::move(t2));            // Compliant
}

void func_auto(A& var)
{
    auto&& var1 = var;
    f1(std::move(var1));                    // Non-compliant
    f1(std::forward<decltype(var1)>(var1)); //Compliant
}

void main()
{
    int32_t i;
    func(0, i);
}
```

In this example, template function `func` forwards parameters `t1` and `t2` to function `f1`. Polyspace flags the use of `std::forward` with `t2` because this parameter is an rvalue reference (type T&).

Polyspace also flags the use of `std::move` with `t1` because this parameter is a forwarding reference (type T&&). If `t1` is initialized with an lvalue, the move might result in an unexpected modification of the parameter. Similarly, Polyspace flags the use of `std::move` in `func_auto` because objects of type `auto&&` are considered as forwarding references.

## Check Information

**Group:** Language support library
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-9-3

The std::move shall not be used on objects declared const or const&

## Description

### Rule Definition

*The std::move shall not be used on objects declared const or const&.*

### Rationale

When you use `std::move()` on an object, it is cast into an rvalue. The compiler then manages the resources in the object by calling the constructor or operator with the closest matching parameter list. If you call `std::move()` on a `const` or `const&` type object, the call returns a `const` or `const&` type rvalue. Because move constructors and operators do not take a `const` type argument, the compiler calls the copy constructor or operator instead of the move constructor or operator. Consider this code snippet where a `const` object is copied when you might expect a move after a call to `std::move()`.

```
class string{
    //...
public:
    string(const string& rhs);// copy contructor
    string(string&& rhs);     //move constructor
};

void print(string text) {
    cout<<text;
    //...
}

int main(){
    int const message = "Error";
    //..
    print(std::move(message))// the copy constructor is called
}
```

The return type of `std::move(message)` is the rvalue `const string&&`. Between the move and copy constructors of class `string`, only the copy constructor accepts `const` type argument. The compiler calls the copy constructor and copies the resources of `message` into `text`.

Because `std::move()` does not move a `const` or `const&` type object, avoid using `std::move()` on `const` or `const&` objects. If you intend to move resources from an object, do not declare it as `const` or `const&`.

### Polyspace Implementation

Polyspace flags use of `std::move()` on:

- Objects that are declared `const` or `const&`.
- Objects that are cast to `const` or `const&`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using `std::move()` on `const` and `const&` Objects

```
#include <cstdint>
#include <utility>
class A
{
    // Implementation
};
void F1(const int32_t &is_const, int32_t &is_non_const)
{
    const A a1{};
    int32_t target = 0;
    A a2 = a1;              // Compliant
    A a3 = std::move(a1);   // Noncompliant

    target =
    std::move((const int32_t &)is_non_const);// Noncompliant
    target =
    std::move(static_cast<const int32_t &>(is_non_const));// Noncompliant
    target =
    std::move(const_cast<int32_t &>(is_const));// Compliant
}
int main(){
    //...
}
```

- Polyspace flags the use of `std::move()` with `const` object `a1`. The compiler calls the copy constructor to copy `a1` to `a3`. You might expect the compiler to call the move constructor.
- Polyspace also flags the use of `std::move()` with the object `is_non_const` when it is cast to `const`. After the casting, the compiler calls the copy constructor to copy `is_non_const` to `target`. You might expect the compiler to call the move constructor.
- Polyspace does not flag the use of `std::move()`with the non-`const` object that results from casting the `const` object `is_const` into a non-`const` type by using `const_cast`. After casting, `is_const` is no longer a `const` object. The compiler calls the move constructor.

## Check Information

**Group:** Language support library
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A18-9-4

An argument to std::forward shall not be subsequently used

## Description

### Rule Definition

*An argument to std::forward shall not be subsequently used.*

### Rationale

You typically use `std::forward` in a function template to pass a forwarding reference parameter to another function. The resources of the parameter might be transferred to another object through a move operation, depending on the value category of the parameter.

For an rvalue parameter, the parameter is in an indeterminate state if it is moved from after the call to `std::forward` and it should not be reused.

For an lvalue parameter, If you reuse the parameter after the call to `std::forward`, modifications to the parameter might affect the argument of the caller function to which you pass the parameter.

### Polyspace Implementation

Polyspace flags the call to `std::forward` if the forwarded object is reused after the call. Polyspace also highlights the lines where the forwarded object is reused in your code.

Polyspace does not flag the call to `std::forward` if its argument is reused in a branch that cannot be reached after the call to `std::forward`. For instance, in this code snippet, the branch where the reuse of variable `t` occurs cannot be reached after the code enters the branch where `std::forward` is used.

```
template<typename T>
void func(T&& t)
{
  T&& p = t;

  switch(t) {
    case 0:
      p = std::forward<T>(t);
      break;
    case 1:
      t--; //t reused
      break;
  }
}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Reuse of Parameter After Call to `std::forward`

```
#include <cstdint>
#include <iostream>
#include <utility>

namespace myTemplates
{

template <typename ...T>
void f1(T...t);

template<typename T>
void f2(T&& t2, bool b)
{
    if (b) {
        f1(std::forward<T>(t2)); // Compliant
    } else {
        t2++;  // else branch not entered
    }

}


template<typename T>
void f3(T&& t3)
{
    T&& p = std::forward<T>(t3); // Non-compliant

    switch (t3) {                // t3 reused
    case 0:
        t3++;                    // t3 reused
        break;
    case 1:
        t3--;                    // t3 reused
        break;
    default:
        break;
    }
}


template<typename T>
void f4(T&& t4)
{
    --t4;

    f1(std::forward<T>(t4)); // Non-compliant

    t4++;                    // t4 reused

    f1(std::forward<T>(t4)); // Non-compliant and t4 reused

    t4--;                    // t4 reused
}
template<typename T>
void f5(T&& t5)
{
    f1(t5,                   // t5 reused
       std::forward<T>(t5)); // Non-compliant
}

}

void main(void)
{
    int i;

    myTemplates::f2(i, true);
    myTemplates::f3(i);
    myTemplates::f4(i);
    myTemplates::f5(i);

}
```

In this example, Polyspace flags all the calls to `std::forward` where the forwarded parameter is reused after the call. In template function `f4`, the second call to `std::forward` counts as a reuse of the parameter `t4`. There are no violations of this rule in `f2` because `t2` is reused in the `else` branch which is never entered.

## Check Information

**Group:** Language support library
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A20-8-1

An already-owned pointer value shall not be stored in an unrelated smart pointer

## Description

### Rule Definition

*An already-owned pointer value shall not be stored in an unrelated smart pointer.*

### Rationale

You use smart pointers to ensure that the memory a pointer points to is automatically deallocated when the pointer is destroyed, for example if the pointer goes out of scope. When unrelated smart pointers manage the same pointer value, one of the smart pointers might attempt to deallocate memory that was already deallocated by the other smart pointer. This results in a double free vulnerability, which corrupts your program's memory management data structure.

A smart pointer owns the pointer value that is used to initialize the smart pointer. If a pointer value is already owned by a smart pointer such as `std::shared_ptr`, and then you use that smart pointer to initialize another smart pointer, for example with a copy operation, the two smart pointers are related. The underlying pointer value is managed by both smart pointers and the memory pointed to is not deallocated until all the smart pointers are destroyed.

### Polyspace Implementation

Polyspace flags the use of an already-owned pointer as the argument of:

- A smart pointer constructor. For instance, in this code snippet, `raw_ptr` is already owned by `s_ptr1` and is used to initialize `s_ptr2`:

  ```
  char *raw_ptr = new char;
  std::shared_ptr<char> s_ptr1(raw_ptr);
  std::shared_ptr<char> s_ptr2(raw_ptr); //raw_ptr is already owned by s_ptr1
  ```
- A smart pointer reset operation. For instance, in this code snippet, the reset of `s_ptr2` replaces `raw_ptr2` with already-owned `raw_ptr1`:

  ```
  char *raw_ptr1 = new char;
  char *raw_ptr2 = new char;

  std::shared_ptr<char> s_ptr1(raw_ptr1);
  std::shared_ptr<char> s_ptr2(raw_ptr2);

  s_ptr2.reset(raw_ptr1); // s_ptr2 releases raw_ptr2 and owns already owned raw_ptr1
  ```

Polyspace checks only smart pointer types `std::shared_ptr` and `std::unique_ptr` and considers that user-defined allocators and deleters have standard allocation and deallocation behavior.

A pointer is already owned by a smart pointer if the pointer type is not `std::nullptr_t` and either:

- The pointer was used to initialize the smart pointer.
- The pointer was used as an argument to the smart pointer `reset()` member function.
- The pointer is the return value of the smart pointer `get()` member function.
- The pointer is the return value of the smart pointer `operator->` member function.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of an Already-Owned Pointer

```
#include <memory>
#include <string>

struct Profile
{
    virtual ~Profile()=default;
};

struct Player : public Profile
{
    std::string name;
    std::int8_t rank;

    Player();
    Player(const std::string& name_, const std::int8_t& rank_) :
        name{ name_ }, rank{ rank_ } {}
};

void func(){

    Player * player = new Player("Richard Roll",1);
    std::shared_ptr<Player> player1(player);
    std::shared_ptr<Player> top_rank(player); //Non-compliant

}

void func2(){

    std::shared_ptr<Player> player1_shared =
        std::make_shared<Player>("Richard Roll",1);
    std::shared_ptr<Player> top_rank_shared(player1_shared); //Compliant

}
```

In this example, the use of pointer value `player` to construct smart pointer `top_rank` in function `func` is non-compliant. `player` is already owned by smart pointer `player1`. When `player1` is destroyed, it might attempt to delete pointer value `player` which was already deleted by `top_rank`.

If you intend to have multiple smart pointer manage the same pointer value, use `std::make_shared` to declare `player1_shared`, and then use copy construction to create related smart pointer `top_rank_shared`, as in `func2`. The underlying pointer value is not deleted until all smart pointers are destroyed.

If you do not intend to share the pointer value between smart pointers, use `std::make_unique` to construct a smart pointer of type `std::unique_ptr`. A `std::unique_ptr` can only be moved, which relinquishes ownership of the underlying managed pointer value.

## Check Information

**Group:** General utilities library
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `CERT C++: MEM56-CPP`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A20-8-2

A std::unique_ptr shall be used to represent exclusive ownership

## Description

### Rule Definition

*A std::unique_ptr shall be used to represent exclusive ownership.*

### Rationale

Raw pointers to heap memory suffer from two related problems:

- When a raw pointer goes out of scope, the pointed memory might not be deallocated and result in a memory leak. You have to remember to explicitly deallocate the memory (`delete` the pointer) before the pointer goes out of scope.
- If you pass a raw pointer to a function, it is unclear if the function takes exclusive ownership of the pointed resource and can deallocate the memory or must leave the deallocation to the caller. If the function deallocates the memory, there is a risk that another pointer pointing to the same memory location is now left dangling.

A `std::unique_ptr` object is a smart pointer that solves both problems and does not require significant additional overheads over raw pointers:

- You do not have to explicitly deallocate the pointed memory. The memory is deallocated before the pointer goes out of scope.
- The pointer has exclusive ownership of the pointed object. When you pass the pointer to a function by a move operation, the function assumes ownership of the memory through the pointer and implicitly deallocates the memory on completion (unless you pass the ownership to another function).

### Polyspace Implementation

The checker flags functions other than `main` that have raw pointers as parameters or return values.

The checker raises a violation of both this rule and `AUTOSAR C++14 Rule A20-8-3`.

- If you want the function to take exclusive ownership of the pointed object, convert the raw pointer to `std::unique_ptr` type.
- If you want the function to take shared ownership of the pointed object, convert the raw pointer to `std::shared_ptr` type.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Raw Pointers as Function Parameters

```
#include <memory>
#include <cstdint>
#include <thread>
constexpr std::uint32_t SIZE=100;

class Resource {
  public:
    bool lookup(std::int32_t);
  private:
    std::int32_t arr[SIZE];
};

bool doesValueExist(Resource *aResource, std::int32_t val) { //Noncompliant
    return aResource->lookup(val);
}

bool doAllSmallerValuesExist(std::unique_ptr<Resource> aResource, std::int32_t val) {
//Compliant
    bool valueExists = true;
    for(std::int32_t i = 0; i <= val; i++) {
        valueExists = aResource->lookup(i);
        if(!valueExists)
            break;
    }
    return valueExists;
}

std::int32_t getAVal();

void main(void) {
    Resource *aResourcePtr = new Resource;
    auto anotherResourcePtr = std::make_unique<Resource>();
    bool valueFound, allSmallerValuesFound;

    //Initialize resources

    valueFound = doesValueExist(aResourcePtr, getAVal());
    allSmallerValuesFound = doAllSmallerValuesExist(std::move(anotherResourcePtr), getAVal());
}
```

In this example, the function `doesValueExist` takes a raw pointer to a `Resource` object as parameter and violates the rule.

The function `doAllSmallerValuesExist` performs similar operations on a `Resource` object but takes an `std::unique_ptr` pointer to the object as parameter.

## Check Information

**Group:** General utilities library
**Category:** Required, Automated

## Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A20-8-3

A std::shared_ptr shall be used to represent shared ownership

## Description

### Rule Definition

*A std::shared_ptr shall be used to represent shared ownership.*

### Rationale

Raw pointers to heap memory suffer from two related problems:

- When a raw pointer goes out of scope, the pointed memory might not be deallocated and result in a memory leak. You have to remember to explicitly deallocate the memory (`delete` the pointer) before the pointer goes out of scope.
- If you pass a raw pointer to a function, it is unclear if the function takes exclusive ownership of the pointed resource and can deallocate the memory or must leave the deallocation to the caller. If the function deallocates the memory, there is a risk that another pointer pointing to the same memory location is now left dangling.

A `std::shared_ptr` object is a smart pointer that solves both problems.

- You do not have to explicitly deallocate the pointed memory. The memory is deallocated before the last pointer pointing to the memory location goes out of scope.
- The pointer has shared ownership of the pointed object. When you pass the pointer to a function, the function assumes ownership of the memory through the pointer and implicitly deallocates the memory on completion as long as no other pointer is pointing to the object.

Although a `std::shared_ptr` object has some overhead over a raw pointer, the use of this object avoids possible memory leaks later.

### Polyspace Implementation

The checker flags functions other than `main` that have raw pointers as parameters or return values.

The checker raises a violation of both this rule and `AUTOSAR C++14 Rule A20-8-2`.

- If you want the function to take exclusive ownership of the pointed object, convert the raw pointer to `std::unique_ptr` type.
- If you want the function to take shared ownership of the pointed object, convert the raw pointer to `std::shared_ptr` type.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Raw Pointers as Function Parameters

```
#include <memory>
#include <cstdint>
#include <thread>
constexpr std::uint32_t SIZE=100;

class Resource {
    public:
       bool lookup(std::int32_t);
    private:
       std::int32_t arr[SIZE];
};

bool doesValueExist(Resource *aResource, std::int32_t val) { //Noncompliant
    return aResource->lookup(val);
}

bool doAllSmallerValuesExist(std::shared_ptr<Resource> aResource, std::int32_t val) {
//Compliant
    bool valueExists = true;
    for(std::int32_t i = 0; i <= val; i++) {
        valueExists = aResource->lookup(i);
        if(!valueExists)
            break;
    }
    return valueExists;
}

std::int32_t getAVal();

void main(void) {
    Resource *aResourcePtr = new Resource;
    auto anotherResourcePtr = std::make_shared<Resource>();
    bool valueFound, allSmallerValuesFound;

    //Initialize resources

    valueFound = doesValueExist(aResourcePtr, getAVal());
    allSmallerValuesFound = doAllSmallerValuesExist(anotherResourcePtr, getAVal());
}
```

In this example, the function `doesValueExist` takes a raw pointer to a `Resource` object as parameter and violates the rule.

The function `doAllSmallerValuesExist` performs similar operations on a `Resource` object but takes an `std::shared_ptr` pointer to the object as parameter.

## Check Information

**Group:** General utilities library
**Category:** Required, Automated

## Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A20-8-4

A std::unique_ptr shall be used over std::shared_ptr if ownership sharing is not required

## Description

### Rule Definition

*A std::unique_ptr shall be used over std::shared_ptr if ownership sharing is not required.*

### Rationale

A resource can be owned by a single `std::unique_ptr` object. In contrast, a resource can be shared by several `std::shared_ptr` objects. Using `std::unique_ptr` when resource sharing is not required has these advantages:

- An `std::shared_ptr` object keeps an internal count of objects that share a resource. Copy assigning or copy constructing an `std::shared_ptr` object increments this use count. An `std::unique_ptr` does not keep this count, making it a more efficient alternative.
- The resource owned by an `std::unique_ptr` object has a predictable life cycle. Unless its ownership is moved to a different `std::unique_ptr` object, the resource is released when the object goes out of scope. The resource managed by an `std::shared_ptr` object has an unpredictable life cycle. It is not released until all the objects that share it are out of scope.

Using `std::shared_ptr` objects makes your code inefficient and difficult to debug. Avoid using `std::shared_ptr` objects unless ownership sharing is required. Instead, use `std::unique_ptr` objects as smart pointers.

### Polyspace Implementation

Polyspace raises a violation of this rule when replacing a `std::shared_ptr` object in your code by an `std::unique_ptr` object is possible. For instance, a violation is raised when either of these conditions are true:

- An `std::shared_ptr` is not used in a copy construction or copy assignment. The internal reference count remains at one. Consider this code:

  ```
  #include <memory>
  class A();
  void bar(std::shared_ptr<A> sp);
  void foo(){
  auto sp = std::make_shared<A>();//count==1
  bar(sp);
  }
  ```

  No `std::shared_ptr` objects are copy constructed or copy assigned. The object `sp` does not share its resources. Declaring this object as an `std::unique_ptr` might be more efficient. See “Avoid Using shared_ptr Objects If They are Not Copied” on page 26-544.
- `std::shared_ptr` objects are sequentially copied to create a chain of `N` shared pointers, but the first `(N-1)` shared pointers are not dereferenced or passed to a function. Because these `(N-1)` shared pointers are not used, moving their resources and declaring the pointers as unique pointers might be more efficient. For instance, in this code:

```
#include <memory>
class obj();
void foo(){
    auto A = std::make_shared<obj>();
    auto B = A;//count==2
    auto C = B;//count==3
    //…
    auto X = *C;
}
```

A chain of shared pointers is created by copying A into B and then copying B into C. Because A and B are not dereferenced, declaring A, B, and C as `std::unique_ptr` objects does not result in loss of functionality and makes the code more efficient. See "Avoid Using shared_ptr Objects When They Are Copied Once But Not Dereferenced" on page 26-545.

As an exception, Polyspace does not raise a violation if you use an `std::shared_ptr` object as an argument of certain standard atomic operation.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Using shared_ptr Objects If They are Not Copied

```
#include <memory>
#include <cstdint>
#include <thread>

struct A {
    A(std::uint8_t xx, std::uint8_t yy) : x(xx), y(yy) {}
    std::uint8_t x;
    std::uint8_t y;
};
void Bar(std::shared_ptr<A> obj) {/**/ }

void Func1()
{
    auto spA = std::make_shared<A>(3,5); //Noncompliant
}

void Func2(void)
{
    std::shared_ptr<A> spA = std::make_shared<A>(3,5);
    //Compliant, object accesses in parallel
    std::thread th1{&Bar, spA};
    std::thread th2{&Bar, spA};
    th1.join();
    th2.join();
}
```

In this example:

- An `std::shared_ptr` object `spA` is declared in `Func1()` but it does not share its resources. Polyspace flags `spA` as noncompliant. The best practice is to declare such pointers as `std::unique_ptr` objects.
- An `std::shared_ptr` object `spA` is declared in `Func2()` and then used in two different threads simultaneously. Polyspace does not raise a violation because `spA` is used in multiple threads at the same time.

### Avoid Using shared_ptr Objects When They Are Copied Once But Not Dereferenced

```
#include <memory>
#include <cstdint>
#include <thread>
#include<iostream>

class Resource {
public:
    Resource(){}
};
class Interface{
public:
    Interface(std::shared_ptr<Resource> obj):R(obj){}
    std::shared_ptr<Resource> R;
};
void foo(){
    auto user1 = std::make_shared<Resource>();//Noncompliant
    auto user2 = user1;
    auto user3 = user2;
    Interface ti1(user3);
}
void foo2(){
    auto user1 = std::make_shared<Resource>();//Compliant
    auto user2 = user1;
    auto user3 = user2;
    //...
    auto X = *user2;
}
void bar(){
    auto user1 = std::make_unique<Resource>();
    auto user2 = std::move(user1);
    auto user3 = std::move(user2);
    //Interface ti1(user3);
}
```

In this example, a cheap interface class `Inteface` contains a pointer to an expensive object of class `Resource`.In `foo()`, three shared pointers `user1`, `user2`, and '`user3` are created by copying `user1` into `user2` and then copying `user2` into `user3`. The developer intent might be to create three separate `Interface` objects by using these shared pointers. Perhaps inadvertently, only one `Interface` object is created by using `user3`. Because `user1` and `user2` do not access their resource, it is possible to move the resource from one pointer to the other until `user3` owns the resource. It might be more efficient to replace these shared pointers in by unique pointers, as shown in `bar()`. Polyspace raises a violation on `foo()`.

The function `foo2()` contains a similar chain of copied shared pointers. Because `user2` accesses its resources, moving its resources to `user3` results in loss of functionality. In this case, it is not possible to replace the shared pointers by unique pointers. Polyspace does not raise a violation.

### Use Temporary Objects to Avoid Creating shared_ptr Objects That Do Not Share Resources

```
#include <memory>
class A{A();};
extern void foo(std::shared_ptr<A> sp);
void func1(){
    auto var = std::make_shared<A>();//Noncompliant
    foo(var);
    //...
}

void func2(){
    foo(std::make_shared<A>());//Compliant
    //...
}

void func3(){
    auto up = std::make_unique<A>();
    foo (std::shared_ptr<A>(std::move(up)));//Compliant
}
```

In this example, the external function `foo()` accepts an `std::shared_ptr` as input. In `func1`, to call `foo()`, the `std::shared_ptr` object `var` is created, but this object does not share ownership of its resources. Polyspace raises a violation.

To resolve this defect, modify `foo()` to accept a `unique_ptr`. If modifying `foo()` is not feasible, there are two alternative solutions.

- You might create a temporary `std::shared_ptr` object and pass it directly to `foo()`, as shown in `func2`.
- You might create a `std::unique_ptr` object and then convert it to a `std::shared_ptr` object when passing it to `foo()`, as shown in`func3`.

## Check Information

**Group:** General utilities library
**Category:** Required, Automated

## Version History

**Introduced in R2022b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A20-8-5

std::make_unique shall be used to construct objects owned by std::unique_ptr

## Description

### Rule Definition

*std::make_unique shall be used to construct objects owned by std::unique_ptr.*

### Rationale

Instead of allocating memory by using the `new` operator and converting the resulting raw pointer to an `std::unique_ptr` object, for instance:

```
class numberClass {
   public:
     numberClass(int n): number(n){}
   private:
     int number;
}
int aNumber=1;
std::unique_ptr<numberClass> numberPtr (new numberClass(aNumber));
```

Create a `std::unique_ptr` object directly using the `std::make_unique` function. For instance:

```
auto numberPtr = std::make_unique<numberClass>(aNumber);
```

Using `std::make_unique` is preferred because:

- The creation of the `std::unique_ptr` object using `std::make_unique` is exception-safe. Otherwise, an exception can occur between the dynamic memory allocation with the `new` operator and the subsequent conversion, leading to a memory leak. An exception causes a memory leak only in certain contexts, for instance, when the `std::unique_ptr` object is created in an argument of a multi-parameter function and another function argument evaluation throws an exception.
- You can use a more concise syntax. You do not have to repeat the data type of the object that is dynamically allocated.

### Polyspace Implementation

The checker flags the creation of an `std::unique_ptr` object (or `boost::unique_ptr` object) from the raw pointer returned by the `new` operator.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** General utilities library
**Category:** Required, Automated

## Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A20-8-6

std::make_shared shall be used to construct objects owned by std::shared_ptr

## Description

### Rule Definition

*std::make_shared shall be used to construct objects owned by std::shared_ptr.*

### Rationale

Instead of allocating memory by using the `new` operator and converting the resulting raw pointer to an `std::shared_ptr` object, for instance:

```
class numberClass {
   public:
     numberClass(int n): number(n){}
   private:
     int number;
}
int aNumber=1;
std::shared_ptr<numberClass> numberPtr (new numberClass(aNumber));
```

Create a `std::shared_ptr` object directly using the use `std::make_shared` function. For instance:

```
auto numberPtr = std::make_shared<numberClass>(aNumber);
```

Using `std::make_shared` is preferred because:

- The creation of the `std::shared_ptr` object is performed in a single dynamic memory allocation and improves run-time performance. Otherwise, creating a raw pointer by using the `new` operator requires one dynamic memory allocation and converting the raw pointer to an `std::shared_ptr` object requires a second allocation. The second allocation creates a control block that keeps track of the reference count of the shared resource and makes the `std::shared_ptr` object aware of all pointers to the shared resource.
- The creation of the `std::shared_ptr` object using `std::make_shared` is exception-safe. Otherwise, an exception can occur between the dynamic memory allocation with the `new` operator and the subsequent conversion, leading to a memory leak. An exception causes a memory leak only in certain contexts, for instance, when the `std::shared_ptr` object is created in an argument of a multi-parameter function and another function argument evaluation throws an exception.
- You can use a more concise syntax. You do not have to repeat the data type of the object that is dynamically allocated.

### Polyspace Implementation

The checker flags the creation of an `std::shared_ptr` object (or `boost::shared_ptr` object) from the raw pointer returned by the `new` operator.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** General utilities library
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A20-8-7

A std::weak_ptr shall be used to represent temporary shared ownership.

## Description

### Rule Definition

*A std::weak_ptr shall be used to represent temporary shared ownership.*

### Rationale

A `std::shared_ptr` is deallocated when the reference count drops to zero. If the code creates a reference cycle by using multiple `std::shared_ptr`, the reference count of any `std::shared_ptr` involved in the cycle can never drop to zero. This can cause a memory leak.

Use `std::weak_ptr` to break up reference cycles. A `std::weak_ptr` can point to a `std::shared_ptr` but does not increase the reference count.

### Polyspace Implementation

Polyspace raises this defect when assigning by using the assignment operator of `std::shared_ptr`. If the right side is a class or struct field of type `std::shared_ptr` that is instantiated with a template that has only a `std::shared_ptr` template argument, Polyspace flags this defect.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Potential Reference Cycles When Using std::shared_ptr

```
#include <memory>

template <template <typename> class T, typename U>
struct Base {
  T<U> sPoint;
};

template <typename T>
using Ex1 = Base<std::shared_ptr, T>;

struct spExampleA;
struct spExampleB : public Ex1<spExampleA> {};
struct spExampleA : public Ex1<spExampleB> {};


void example()
{
  std::shared_ptr<spExampleB> sp1 = std::make_shared<spExampleB>();
  std::shared_ptr<spExampleA> sp2 = std::make_shared<spExampleA>();
  sp1->sPoint = sp2;             // Noncompliant
  sp2->sPoint = sp1;             // Noncompliant

}
```

The preceding code creates a reference cycle with `std::shared_ptr`. In the example, neither `sp1` nor `ps2` can have their reference count drop to 0 and be deallocated. Use `std::weak_ptr` instead of `std::shared_ptr` to break reference cycles.

## Check Information

**Group:** General utilities library
**Category:** Required, Non-automated

# Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A21-8-1

Arguments to character-handling functions shall be representable as an unsigned char

## Description

### Rule Definition

*Arguments to character-handling functions shall be representable as an unsigned char.*

### Rationale

You cannot use plain `char` variables as arguments to character-handling functions declared in `<cctype>`, for instance, `isalpha()` or `isdigit()`. On certain platforms, plain `char` variables can have negative values that cannot be represented as `unsigned char` or EOF, resulting in undefined behavior.

### Polyspace Implementation

The check raises a flag when you use a signed or plain `char` variable with a negative value as argument to a character-handling function.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Strings library
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A23-0-1

An iterator shall not be implicitly converted to const_iterator

## Description

### Rule Definition

*An iterator shall not be implicitly converted to const_iterator.*

### Rationale

The C++11 standard introduces member functions such as `cbegin` and `cend` that returns const iterators to containers. To create const iterators, use these member functions instead of functions such as `begin` and `end` that return non-const iterators and then require implicit conversions.

For instance, consider the `std::list` container:

```
std::list<int> aList = {0, 0, 1, 2};
```

You can use the `begin` and `end` member functions of the container to create const iterators, for instance in a `for` loop:

```
for(std::vector<int>::const_iterator iter{aList.begin()}, end{aList.end()};
    iter != end;
    ++iter) {...}
```

However, the functions `begin` and `end` return non-const iterators and for assignment to the const iterators `iter` and `end` respectively, an implicit conversion must happen. Instead, take advantage of the new C++11 functions `cbegin` and `cend` that directly returns const iterators:

```
for(std::vector<int>::const_iterator iter{aList.cbegin()}, end{aList.cend()};
    iter != end;
    ++iter) {...}
```

If you use these functions, you can also replace the explicit type specification of the iterators with `auto`:

```
for(auto iter{aList.cbegin()}, end{aList.cend()};
    iter != end;
    ++iter) {...}
```

### Polyspace Implementation

The checker flags conversions from type `iterator` to `const_iterator` or `reverse_iterator` to `const_reverse_iterator`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Implicit Conversions to Const Iterators

```
#include <cstdint>
#include <vector>

void func(std::vector<int32_t> & values, int32_t aValue) {
    std::vector<int32_t>::const_iterator iter1 =
                 std::find(values.begin(), values.end(), aValue); //Noncompliant
    std::vector<int32_t>::const_iterator iter2 =
                 std::find(values.cbegin(), values.cend(), aValue);  //Compliant
}
```

In this example, the first `std::find` function call uses as arguments the return values of the `begin` and `end` methods of an `std::vector` container `values`. These methods return iterators of type `std::vector<intr32_t>::iterator`. Since the `std::find` template has the same return type as the types of the first two arguments, it also returns an iterator of type `std::vector<intr32_t>::iterator`. The return value is assigned to a variable of type `std::vector<intr32_t>::const_iterator`, resulting in an implicit conversion.

The second call uses the `cbegin` and `cend` methods which return iterators of type `std::vector<intr32_t>::const_iterator` and avoid the implicit conversion.

## Check Information

**Group:** Language support library
**Category:** Required, Automated

# Version History

**Introduced in R2020a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A23-0-2

Elements of a container shall only be accessed via valid references, iterators, and pointers

## Description

### Rule Definition

*Elements of a container shall only be accessed via valid references, iterators, and pointers.*

### Rationale

You access the elements of a container by dereferencing a reference, a pointer, or an iterator. When the size or shape of a container changes, the reference, pointer, or iterator might point to a different element or an invalid location. For instance, consider this code:

```
std::vector<int> v = {1,2,3,4,5,6,7};
std::vector<int>::iterator it = std::find(v.begin(), v.end(), 5);;
v.push_back(-1);
std::cout<<*it;
```

If the `push_back()` operation causes the vector v to reallocate memory to accommodate a larger size, the iterator `it` is no longer valid. Dereferencing the invalid iterator might produce unexpected results. If the invalidated iterator becomes a dangling pointer or an uninitialized pointer, you might not be able to dereference it safely.

The C++ standard defined which operations invalidate the iterators of the standard library containers. See Containers library. When performing actions that might invalidate an iterator, validate the iterator, for instance, by recalculating its position.

### Polyspace Implementation

Polyspace raises a violation of the rule when you access the elements of a container by using a pointer, reference, or iterator that has been invalidated.

Certain erasure operations, such as `std::deque::pop_back()` or `std::forward_list::pop_front()` invalidate the iterators to the erased elements. After the erasure, whether any iterators pointed to the erased elements is unknown. If invalidated iterators that point to the erased elements exist in your code, Polyspace does not flag them.

Certain operations return iterators through an `std::pair`. Polyspace does not track iterators that are part of an `std::pair`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using Invalid Iterators

```
#include <iostream>
#include <vector>
```

```
#include <algorithm>
#include <list>
void foo()
{
    std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7};
    std::vector<int>::iterator it = std::find(v.begin(), v.end(), 5);
    v.push_back(8);
    std::cout<<*it; //Noncompliant
}

void bar()
{
    std::list<int> l{0, 1, 2, 3, 4, 5, 6, 7};
    std::list<int>::iterator it = std::find(l.begin(), l.end(), 5);
    l.remove(4);
    std::cout<<*it; //Compliant
}
```

In this example, the function `foo()` dereferences an iterator `it` after an insertion. The insertion might trigger a reallocation and invalidate `it`. Polyspace flags the dereferencing of an iterator that might be invalid.

The function `bar()` dereferences `it` after an erasure. The erasure does not invalidate the iterator and Polyspace does not flag the dereferencing.

## Check Information

**Group:** Language support library
**Category:** Required, Automated

# Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A25-1-1

Non-static data members or captured values of predicate function objects that are state related to this object's identity shall not be copied

## Description

### Rule Definition

*Non-static data members or captured values of predicate function objects that are state related to this object's identity shall not be copied.*

### Rationale

Predicate function objects take a single argument and use that argument to return a value that is testable as a boolean (true or false). Standard Template Library (STL) algorithms that accept a predicate function object may, depending on the algorithm implementation, make a copy of the function object. The invocation of the copy might cause unexpected results.

For example, suppose that you use an algorithm to remove one element from a list. To determine the element to remove, the algorithm uses a predicate function object that modifies a state related to the function object identity each time the function object is called. You might expect the predicate object to return true only once. If the algorithm makes a copy of the predicate object, the state of the copy is also modified and the copy returns true a second time, removing a second element from the list.

To avoid unexpected results, consider one of the following:

- Wrap the predicate function object in a `std::reference_wrapper<T>` before you pass it to the algorithm. If the wrapper object is copied, all copies refer to the same underlying predicate function object.

  For example, in this code snippet, predicate function object `myObj` is wrapped in `std::ref` when passed to `std::remove_if`.

  ```
  class predObj {
      // Defines function object that modifies its state
  };
  void func() {
      std::vector<int> v{0, 1, 2, 3, 4, 5};
      //
      predObj myObj;
      v.erase(std::remove_if(v.begin(), v.end(), std::ref(myObj)), v.end());
      //....
  }
  ```
- Implement a `const` function call operator that does not modify the state of the predicate object. For example, in this code snippet, `predModifies` defines a call operator that modifies `elemPosition` before checking the equality whereas `predDoesNotModify` checks only the equality of `var` without modifying it.

  ```
  class predModifies : public std::unary_function<int, bool>
  {
  public:
  predModifies() : elemPosition(0) {}
  bool operator()(const int &) const { return (++elemPosition) == 3; }
  //call operator modifies elemPosition
  private:
  ```

```
mutable size_t elemPosition;
};

class predDoesNotModify: public std::unary_function<int, bool>
{
public:
bool operator()(const int& var) const { return var == 3; }
//call operator does not modify state of object
};
```

### Polyspace Implementation

Polyspace flags violations of this rule when the following conditions are true:

- You use a STL algorithm that takes a predicate function object. For a full list of STL algorithms, see Algorithms library.
- The state of the function object is modified, where the state is one of the following:
  - The `this` pointer if the predicate object is an instance of a class that defines a function call operator `operator()`.
  - A captured-by-copy value if the predicate object is a lambda function. A lambda function that captures values by reference is compliant.

A function object state is modified if it is written to or if a non-const method is called on it. For example, in this code snippet, non-const method `increment()` modifies the state `elemPosition` of `operator()` :

```
#include <functional>
#include <vector>
#include <algorithm>

class myPred : public std::unary_function<int, bool> {
  public:
    myPred(): elemPosition(0) {}
    void increment() {++elemPosition;} // Non-const method
    bool operator()(const int&) // function call operator
    {
        increment(); // method called on state elemPosition
        return elemPosition == 3;
    }
  private:
    size_t elemPosition;
};

void func(std::vector<int> v) {
    std::remove_if(v.begin(), v.end(), myPred()); // Noncompliant
}
```

Polyspace flags function objects that modify their state even if the function object is not a predicate.

Polyspace does not flag function objects that are wrapped in these:

- `std::ref`
- `std::cref`
- `std::function`

- `std::bind`
- `std::not1`

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Function Object that Modifies Its State

```
#include <functional>
#include <vector>
#include <algorithm>


void lambdaDoesNotModify(std::vector<int> v)
{
  int elemPosition = 0;
  std::remove_if(v.begin(), v.end(),
    [elemPosition](const int & element) mutable {  //Compliant
      return elemPosition == 0 && element == 3;
    });
}

void lambdaCaptureByRef(std::vector<int> v)
{
  int elemPosition = 0;
  std::remove_if(v.begin(), v.end(),
    [&elemPosition](const int & i) { // Compliant
      return ++elemPosition == 3;
    });
}

class nonConstPred : public std::unary_function<int, bool>
{
  public:
    nonConstPred() : elemPosition(0) {}
    void increment() { ++elemPosition; }
    bool operator()(const int &)
    {
      increment();
      return elemPosition == 3;
    }
  private:
    size_t elemPosition;
};

void myFunc(std::vector<int> v)
{
  std::remove_if(v.begin(), v.end(), nonConstPred()); // Noncompliant
}

void myFunc1(std::vector<int> v)
{
  nonConstPred myPred;
  std::remove_if(v.begin(), v.end(), std::ref(myPred)); // Compliant
}
```

In this example, the use of predicate function object `nonConstPred()` inside function `myFunc` is not compliant. The call operator inside class `nonConstPred` uses a non-const method `increment()` that modifies `elemPosition`, the state of the predicate. If the algorithm `std::remove_if` makes a copy of the predicate function object, there might be two instances of `elemPosition == 3` being true, which might cause unexpected results.

The use of the same function object predicate in `myFunc1` is compliant because it is wrapped in `std::ref`. All copies of the wrapper object refer to the same underlying function object and there is only one instance of the state `elemPosition`.

The use of a lambda function as predicate function object in function `lambdaDoesNotModify` is compliant because the state of the function object is not modified. Similarly, the lambda function in `lambdaCaptureByRef` is compliant because the state of the function object is captured by reference.

### Check Information

**Group:** Algorithms library
**Category:** Required, Automated

## Version History

**Introduced in R2022a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A25-4-1

Ordering predicates used with associative containers and STL sorting and related algorithms shall adhere to a strict weak ordering relation

## Description

### Rule Definition

*Ordering predicates used with associative containers and STL sorting and related algorithms shall adhere to a strict weak ordering relation.*

### Rationale

Algorithms and containers of the standard template library (STL) use predicates to sort and compare their elements. The predicates must adhere to strict weak ordering. That is, the predicate must adhere to these requirements:

- Irreflexivity: For all `x`, comparing `x` to itself must always evaluate to `false`.
- Assymetry: For all `x, y:` if comparing `x` to `y` evaluates to `true`, then comparing `y` to `x` must evaluate to `false`.
- Transitivity: For all `x, y, z:` if comparing `x` to `y` and `y` to `z` both evaluate to `true`, then comparing `x` to `z` must also evaluate to `true`.

These compare type methods violate at least one of these requirements:

- `std::less_equal`
- `std::greater_equal`
- `std::equal_to`
- `std::not_equal_to`
- `std::logical_or`
- `std::logical_and`

Using the preceding functions as ordering predicates for algorithms and containers from the STL might result in infinite loops, erratic behavior, and bugs that are difficult to diagnose. Instead, use functions that adhere to the strict weak ordering relation. For instance, use functions such as `std::less` or `std::greater`.

### Polyspace Implementation

Polyspace raises a violation of this rule when you use one of these compare types as predicates in standard library algorithms and containers:

- `std::less_equal`
- `std::greater_equal`
- `std::equal_to`
- `std::not_equal_to`
- `std::logical_or`

- `std::logical_and`

For a list of standard library algorithms that expect a predicate with strict weak ordering, see Compare.

If you use a user-defined predicate function, Polyspace does not check if the custom predicate adheres to strict weak ordering.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using Predicates That Do Not Adhere to Strict Weak Ordering with STL Algorithms

```
#include <functional>
#include <iostream>
#include <set>
#include <map>

int main(void)
{
    // GE
    std::set<int, std::greater_equal<int>>//Noncompliant
        s1{2, 5, 8};
    auto r = s1.equal_range(5);
    //returns 0
    std::cout << std::distance(r.first, r.second) << std::endl;

    //LE
    std::map<int, std::string, std::less_equal<int>>//Noncompliant
        m1{{2, "AB"}, {5, "CD"}, {8, "EF"}};
    auto r3 = m1.equal_range(5);
    //returns 0
    std::cout << std::distance(r3.first, r3.second) << std::endl;

    return 0;
}
```

In this example, Polyspace flags the use of inappropriate predicates when declaring STL containers. For instance:

- The predicate `greater_equal` does not return false when comparing the same objects, violating the irreflexivity requirement.
- The predicate `less_equal` does not return false when comparing the same objects, violating the irreflexivity requirement.

Polyspace flags the use of these function as predicates when declaring containers such as `std::set` or `std::map`.

By default, these containers use the function `std::less` as the ordering predicate. Because this function adheres to strict weak ordering relation, Polyspace does not raise a violation when you declare the containers by using the default predicate.

## Check Information

**Group:** Algorithms library
**Category:** Required, Non-automated

# Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A26-5-1

Pseudorandom numbers shall not be generated using std::rand()

## Description

### Rule Definition

*Pseudorandom numbers shall not be generated using std::rand().*

### Rationale

This cryptographically weak routines is predictable and must not be used for security purposes. When a predictable random value controls the execution flow, your program is vulnerable to malicious attacks.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Random Loop Numbers

```
#include <stdio.h>
#include <stdlib.h>

volatile int rd = 1;
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;

    nloops = rand();

    for (j = 0; j < nloops; j++) {
        if (random_r(&buf, &i))
            exit(1);
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

This example uses `rand` and `random_r` to generate random numbers. If you use these functions for security purposes, these PRNGs can be the source of malicious attacks.

#### Correction — Use Stronger PRNG

One possible correction is to replace the vulnerable PRNG with a stronger random number generator.

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <openssl/rand.h>

volatile int rd = 1;
int main(int argc, char* argv[])
{
    int j, r, nloops;
    unsigned char buf;
    unsigned int seed;
    int i = 0;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <seed> <nloops>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    seed = atoi(argv[1]);
    nloops = atoi(argv[2]);

    for (j = 0; j < nloops; j++) {
        if (RAND_bytes(&buf, i) != 1)
            exit(1);
        printf("RAND_bytes: %u\n", (unsigned)buf);
    }
    return 0;
}
```

## Check Information

**Group:** Algorithms library
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A26-5-2

Random number engines shall not be default-initialized

## Description

### Rule Definition

*Random number engines shall not be default-initialized.*

### Rationale

Pseudorandom number generators depend on an initial seed value to generate a sequence of random numbers. Default initialization of random number engines is done by using a default seed, which is a constant value. If you call a random number generator that has default initialization multiple times, you get the same sequence of random numbers every time. To avoid unexpected program behavior, such as generating the same sequence of random numbers in different program executions, use unique, nondefault seed values each time that you initialize a random number generator.

An exception to this rule is allowed when you might want a deterministic sequence for consistent testing purposes.

### Polyspace Implementation

The checker reports violations on the lines in which:

- A C++ standard random number generator is default-initialized.
- The seeding function of a random number generator is called by using an implicit call to default arguments or an explicit `default_seed` argument..

**Note** The checker does not report random number engine initializations that have constant input arguments.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant Random Number Generator Initialization

The declaration of `std::default_random_engine eng1` is noncompliant because it is constructed by using the default argument, the `default_seed` constant.

```
#include <iostream>
#include <random>

int main()
{
  std::default_random_engine eng1{};     //Noncompliant
  std::uniform_int_distribution<int> ud2{0, 100};
  std::random_device rd;
  std::default_random_engine eng2{rd()}; //Compliant
  std::default_random_engine eng3{rd()}; //Compliant
  eng3.seed();                           //Noncompliant

return 0;
}
```

The second declaration `std::default_random_engine eng2` is compliant because it takes a user-defined `random_device` object as its initialization argument.

The declaration of `std::default_random_engine eng3` is also compliant. `eng3.seed()` is noncompliant because the seeding function `std::default_random_engine seed` uses the `default_seed` constant as an argument, which overwrites the seed of `eng3` that is correctly initialized.

## Check Information

**Group:** Algorithms library
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A27-0-1

Inputs from independent components shall be validated.

## Description

### Rule Definition

*Inputs from independent components shall be validated.*

### Rationale

When inputs from independent components are directly used in the source code, attackers might get complete or partial control of an application buffer. This control enables an attacker to terminate the program, view the content of the stack, access the content of the memory, modify the memory in random places, and execute unwanted code disguised as the program source code.

To prevent such vulnerabilities, validate the input from independent components. This rule applies to inputs received from external sources, such as:

- Inputs received from networks
- Inputs received from other processes and software through interprocess communication (IPC)
- Inputs received from components API

### Polyspace Implementation

Polyspace raises a flag when inputs from independent components are used without validation. The flagged uses include:

- Routines such as `sethostid` (Linux) or `SetComputerName` (Windows) use externally controlled arguments to change the host ID. See `Host change using externally controlled elements`.
- Functions such as `putenv` and `setenv` obtain new environment variable values or from unsecure sources. See `Use of externally controlled environment variable`.
- Functions such as `printf` use a format specifier that is constructed from unsecure sources. See `Tainted string format`.
- Arrays or pointers use an index that is obtained from unsecure sources. See `Array access with tainted index`.
- The program obtains the path to a command from an external unsecure source. See `Command executed from externally controlled path`.
- The program execute a command that is fully or partially constructed from externally controlled input. See `Execution of externally controlled command`.
- The program loads libraries from fixed or externally controlled unsecure paths that can be partially or fully controlled by attackers. See `Library loaded from externally controlled path`.
- A loop uses values obtained from unsecure sources as its boundary. See `Loop bounded with tainted value`.
- Memory allocation functions, such as `calloc` or `malloc`, use a size argument from an unsecure source. See `Size argument to memory function is from an unsecure source`.

- A pointer dereference uses an offset variable from an unknown or unsecure source. See `Pointer dereference with tainted offset`.
- One or both integer operands in a division operation comes from unsecure sources. See `Tainted division operand`.
- One or both integer operands in a remainder operation (`%`) comes from unsecure sources. See `Tainted modulo operand`.
- String manipulation functions that implicitly dereference the string buffer such as `strcpy` or `sprintf` use strings from unsecure sources. See `Tainted NULL or non-null-terminated string`.
- Values from unsecure sources are implicitly or explicitly converted from signed to unsigned values. See `Tainted sign change conversion`.
- The program dereferences a pointer from an unsecure source that might be NULL or point to unknown memory. See `Use of tainted pointer`.

Polyspace considers these inputs as input from an independent component or tainted:

- Volatile objects
- Object that interact with the user
- Objects that interact with the hardware
- Objects that use random numbers or the current date and time

To consider all input from outside the current analysis perimeter as unsecure, use `-consider-analysis-perimeter-as-trust-boundary`. See “Sources of Tainting in a Polyspace Analysis”.

When an input from an independent component is used without validation multiple times in a code, Polyspace flags the first use.

### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Change Host ID from Function Argument

```
#include <unistd.h>
#include <stdlib.h>

void bug_taintedhostid(void) {
    long userhid = strtol(getenv("HID"),NULL,10);
    sethostid(userhid);//Noncompliant
}
```

This example sets a new host ID using the argument passed to the function. Before using the host ID, check the value passed in.

#### Correction — Predefined Host ID

One possible correction is to change the host ID to a predefined ID. This example uses the host argument as a switch variable to choose between the different, predefined host IDs.

```
#include <unistd.h>
#include <stdlib.h>

extern long called_taintedhostid_sanitize(long);
enum { HI0 = 1, HI1, HI2, HI3 };

void taintedhostid(void) {
    long host = strtol(getenv("HID"),NULL,10);
    long hid = 0;
    switch(host) {
        case HI0:
            hid = 0x7f0100;
            break;
        case HI1:
            hid = 0x7f0101;
            break;
        case HI2:
            hid = 0x7f0102;
            break;
        case HI3:
            hid = 0x7f0103;
            break;
        default:
            /* do nothing */
        break;
    }
    if (hid > 0) {
        sethostid(hid);
    }
}
```

### Set Path in Environment

```
#define _XOPEN_SOURCE
#define _GNU_SOURCE
#include "stdlib.h"

void taintedenvvariable(void)
{
    char* path = getenv("APP_PATH");
    putenv(path); //Noncompliant
}
```

In this example, `putenv` changes an environment variable. The path `path` has not been checked to make sure that it is the intended path.

#### Correction — Sanitize Path

One possible correction is to sanitize the path, checking that it matches what you expect.

```
#define _POSIX_C_SOURCE
#include <stdlib.h>
```

```
#include <string.h>

/* Function to sanitize a path */
const char * sanitize_path(const char* str) {
    /* secure allowlist of paths */
    static const char *const authorized_paths[] = {
        "/bin",
        "/usr/bin"
    };
    if (str != NULL) {
        for (int i = 0; i < sizeof(authorized_paths) / sizeof(authorized_paths[0]); i++)
        if (strcmp(authorized_paths[i], str) == 0) {
            return authorized_paths[i];
        }
    }
    return NULL;
}

void taintedenvvariable(void)
{
    const char* path = getenv("APP_PATH");
    path = sanitize_path(path);
    if (path != NULL) {
        if (setenv("PATH", path, /* overwrite = */1) != 0) {
            /* fatal error */
            exit(1);
        }
    }
}
```

### Get Elements from User Input

```
#include <stdio.h>
#include <unistd.h>
#define MAX 40
void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf(userstr);//Noncompliant
}
```

This example prints the input argument `userstr`. The string is unknown. If it contains elements such as `%`, `printf` can interpret `userstr` as a string format instead of a string, causing your program to crash.

#### Correction — Print as String

One possible correction is to print `userstr` explicitly as a string so that there is no ambiguity.

```
#include "stdio.h"
#include <unistd.h>
#define MAX 40

void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf("%.20s", userstr);
}
```

### Use Index to Return Buffer Value

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];//Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The index `num` is obtained from an unsecure source and the function `taintedarrayindex` does not check to see if `num` is inside the range of `tab`.

#### Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

### Executing Path from Environment Variable

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedpathcmd() {
    char cmd[SIZE128] = "";
    char* userpath = getenv("MYAPP_PATH");

    strncpy(cmd, userpath, SIZE100);//Noncompliant
    strcat(cmd, "/ls *");
    /* Launching command */
    system(cmd);//Noncompliant
}
```

This example obtains a path from an environment variable MYAPP_PATH. The path string is tainted. Polyspace flags its use in the strncopy function. system runs a command from the tainted path without checking the value of the path. If the path is not the intended path, your program executes in the wrong location.

**Correction — Use Trusted Path**

One possible correction is to use a list of allowed paths to match against the environment variable path.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

/* Function to sanitize a string */
/* Any defect is localized here */
int sanitize_str(char* s, size_t n) {
    int res = 0;
    /* String is ok if */
    if (s && n>0 && n<SIZE128) {
        /* - string is not null                     */
        /* - string has a positive and limited size */
        s[n-1] = '\0';  /* Add a security \0 char at end of string *///Noncompliant
        /* Tainted pointer detected above, used as "firewall" */
        res = 1;
    }
    return res;
}

/* Authorized path ids */
enum { PATH0=1, PATH1, PATH2 };

void taintedpathcmd() {
    char cmd[SIZE128] = "";

    char* userpathid = getenv("MYAPP_PATH_ID");
    if (sanitize_str(userpathid, SIZE100)) {
        int pathid = atoi(userpathid);


        char path[SIZE128] = "";
        switch(pathid) {
            case PATH0:
                strcpy(path, "/usr/local/my_app0");
                break;
            case PATH1:
                strcpy(path, "/usr/local/my_app1");
                break;
            case PATH2:
                strcpy(path, "/usr/local/my_app2");
                break;
            default:
                /* do nothing */
        break;
        }
        if (strlen(path)>0) {
            strncpy(cmd, path, SIZE100);
            strcat(cmd, "/ls *");
            system(cmd);
        }
    }
}
```

### Call External Command

```
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "unistd.h"
#include "dlfcn.h"
#include "limits.h"
#define MAX 128


void taintedexternalcmd(void)
{
    char* usercmd;
    fgets(usercmd,MAX,stdin);
    char cmd[MAX] = "/usr/bin/cat ";
    strcat(cmd, usercmd);
    system(cmd);//Noncompliant
}
```

This example function calls a command from a user input without checking the command variable.

#### Correction — Use a Predefined Command

One possible correction is to use a `switch` statement to run a predefined command, using the user input as the switch variable.

```
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "unistd.h"
#include "dlfcn.h"
#include "limits.h"

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
enum { CMD0 = 1, CMD1, CMD2 };

void taintedexternalcmd(void)
{
    int usercmd = strtol(getenv("cmd"),NULL,10);
    char cmd[SIZE128] = "/usr/bin/cat ";

    switch(usercmd) {
        case CMD0:
            strcat(cmd, "*.c");
            break;
        case CMD1:
            strcat(cmd, "*.h");
            break;
        case CMD2:
            strcat(cmd, "*.cpp");
            break;
        default:
            strcat(cmd, "*.c");
    }
    system(cmd);
}
```

**Call Custom Library**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void* taintedpathlib() {
    void* libhandle = NULL;
    char lib[SIZE128] = "";
    char* userpath = getenv("LD_LIBRARY_PATH");
    strncpy(lib, userpath, SIZE128);//Noncompliant- userpath is tainted
    strcat(lib, "/libX.so");
    libhandle = dlopen(lib, 0x00001);//Noncompliant
    return libhandle;
}
```

This example loads the library `libX.so` from an environment variable `LD_LIBRARY_PATH`. An attacker can change the library path in this environment variable. The actual library you load could be a different library from the one that you intend.

**Correction — Change and Check Path**

One possible correction is to change how you get the library path and check the path of the library before opening the library. This example receives the path as an input argument but then performs the following checks on the path:

- The function `sanitize_str` protects against possible buffer overflows.
- The function `identified_safe_libX_folder` checks if the path belongs to a list of allowed paths.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <limits.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

/* Use allowlist */
static const char *libX_safe_folder[] = {
  "/usr/",
  "/usr/lib",
  "/lib"
};

/* Return the index if the input is in the allowlist */
int identified_safe_libX_folder(const char* path)
{
  for (int i = 0; i < sizeof(libX_safe_folder) / sizeof(libX_safe_folder[0]); i ++)
    {
      if (strcmp(path, libX_safe_folder[i]) == 0)
      return i;
    }
  return -1;
}

/* Function to sanitize a string */
char *sanitize_str(char* s, size_t n) {
  /* strlen is used here as a kind of firewall for tainted string errors */
  if (strlen(s) > 0 && strlen(s) < n)
    return s;
  else
    return NULL;
}

void* taintedpathlib(char* userpath) {
  void* libhandle = NULL;
  const char *const checked_userpath = sanitize_str(userpath, SIZE128);
  if (checked_userpath != NULL) {
    int index = identified_safe_libX_folder(checked_userpath);
    if (index > 0) {
      char lib[SIZE128] = "";
      strncpy(lib, libX_safe_folder[index], SIZE128);
      strcat(lib, "/libX.so");
      libhandle = dlopen(lib, RTLD_LAZY);
    }
  }
  return libhandle;
}
```

### Loop Boundary From User Input

```
#include<stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int taintedloopboundary(void) {
    int count;
    scanf("%d", &count);
    int res = 0;
    for (int i=0 ; i < count; ++i) {//Noncompliant
        res += i;
    }
    return res;
}
```

In this example, the function uses a user input to loop `count` times. `count` could be any number because the value is not checked before starting the `for` loop.

#### Correction: Clamp Tainted Loop Control

One possible correction is to clamp the tainted loop control. To validate the tainted loop variable `count`, this example limits `count` to a minimum value and a maximum value by using inline functions `min` and `max`. Regardless of the user input, the value of `count` remains within a known range.

```
#include<stdio.h>
#include<algorithm>
#define MIN 50
#define MAX 128
static  inline int max(int a, int b) { return a > b ? a : b;}
static inline int min(int a, int b) { return a < b ? a : b; }

int taintedloopboundary(void) {
    int count;
    scanf("%d", &count);
    int res = 0;
    count = max(MIN, min(count, MAX));
    for (int i=0 ; i<count ; ++i) {
        res += i;
    }
    return res;
}
```

#### Correction — Check Tainted Loop Control

Another possible correction is to check the low bound and the high bound of the tainted loop boundary variable before starting the `for` loop. This example checks the low and high bounds of `count` and executes the loop only when `count` is between 0 and 127.

```
#include<stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};


int taintedloopboundary(void) {
    int count;
    scanf("%d", &count);
    int res = 0;

    if (count>=0 && count<SIZE128) {
        for (int i=0 ; i<count ; ++i) {
            res += i;
        }
    }
    return res;
}
```

### Allocate Memory Using Input From User

```
#include<stdio.h>
#include <stdlib.h>

int* bug_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = (int*)malloc(size);//Noncompliant
    return p;
}
```

In this example, `malloc` allocates `size` bytes of memory for the pointer `p`. The variable `size` comes from the user of the program. Its value is not checked, and it could be larger than the amount of available memory. If `size` is larger than the number of available bytes, your program could crash.

#### Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the `malloc` operation. This example checks to see if `size` is positive and less than the maximum size.

```
#include<stdio.h>
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int* corrected_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = NULL;
    if (size>0 && size<SIZE128) {          /* Fix: Check entry range before use */
        p = (int*)malloc((unsigned int)size);
    }
    return p;
}
```

### Dereference Pointer Array

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset];//Noncompliant
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of `offset`. Continue with the pointer dereferencing only if `offset` is inside the valid range.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

### Division of Function Arguments

```
#include <limits.h>
#include <stdio.h>

extern void print_int(int);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    int r =  num/denum; //Noncompliant
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

#### Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include <limits.h>
#include <stdio.h>

extern void print_long(long);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    long res= 0;
    if (denum!=0 && !(num==INT_MIN && denum==-1)) {
        res = num/denum;
    }
    print_long(res);
    return res;
}
```

### Modulo with User Input

```
#include <stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem =  128%userden; //Noncompliant
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using a user input. The input is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

#### Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
#include<stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem = 0;
    if (userden > 0 ) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

### Getting String from Input

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));//Noncompliant
    print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

#### Correction — Validate the Data

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // Noncompliant-TAINTED_STRING only flagged here
        // - string is not null
        // - string has a positive and limited size
        // - TAINTED_STRING on strlen used as a firewall
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

**Correction — Validate the Data**

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

### Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size); //Noncompliant
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

#### Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

### Function That Dereferences an External Pointer

```
#include<stdlib.h>
void taintedptr(void) {
    char *p = getenv("ARG");
    char x = *(p+10);//Noncompliant
}
```

In this example, the pointer `*p` points to an string of unknown size. During the dereferencing operation, the pointer might be null or point to unknown memory, which can result in segmentation fault.

#### Correction — Check Pointer

One possible correction is to sanitize the pointer before using it. This example checks whether the pointer is `nullptr` before it is dereferenced.

```
#include<stdlib.h>
#include <string.h>
void taintedptr(void) {
    char *p = getenv("ARG");
    if(p!=nullptr && strlen(p)>10)
    {
    char x = *(p+10);
    }
}
```

## Check Information

**Group:** Input/output library
**Category:** Required, Non-automated

# Version History

**Introduced in R2021b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A27-0-2

A C-style string shall guarantee sufficient space for data and the null terminator

## Description

### Rule Definition

*A C-style string shall guarantee sufficient space for data and the null terminator.*

### Rationale

C-style strings not only require space for the character data written but require one explicit character at the end for the additional null terminator. Failure to accommodate for the space required causes buffer overflow, leading to memory corruption, security vulnerabilities, and other issues.

### Polyspace Implementation

The checker looks for these issues:

- Use of a dangerous standard function.

  This issue occurs when you use C functions such as `gets` and `strcpy`, which write data to a buffer but do not inherently provide controls on the length of data written.

  For a more complete list of functions and their safer alternatives, see `Use of dangerous standard function`.
- Buffer overflow from incorrect string format specifier.

  This issue occurs when the format specifier argument for C functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.
- Destination buffer overflow in string manipulation.

  This issue occurs when certain C string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

  For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.
- Insufficient destination buffer size

  This occurs when the destination buffer in a `strcpy` operation cannot accommodate the source buffer and a null terminator. The issue is reported if the size of the source buffer is unknown.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of a dangerous standard function

```
#include <cstdio>
#include <cstring>

#define BUFF_SIZE 128


int noncompliant_func(char *str) {
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1) { //Noncompliant
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}

int compliant_func(char *str) {
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1) { //Compliant
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

In this example, the rule is violated when you use the `sprintf` function, which does not allow control on the length of data written. To avoid the possible buffer overflow, use the safer alternative function `snprintf`.

### Buffer overflow from incorrect string format specifier

```
#include <cstdio>

void noncompliant_func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf); //Noncompliant
}

void compliant_func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf); //Compliant
}
```

In this example, the buffer `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow and violates the rule. To avoid the rule violation, read a smaller number of elements into the buffer.

### Destination buffer overflow in string manipulation

```
#include <cstdio>

void noncompliant_func(void) {
    char buf[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buf, fmt_string); //Noncompliant
}

void compliant_func(void) {
    char buf[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buf, 20, fmt_string); //Compliant
}
```

In this example, the buffer `buf` can contain 20 `char` elements. Therefore, the greater size of `fmt_string` causes a buffer overflow and violates the rule. To avoid the rule violation, use `snprintf` to enforce length control and read fewer than 20 elements into the buffer.

### Destination Buffer Too Small

In this example, the size of the `source` buffer is unknown, while the size of the `destination` buffer is fixed at `128`. The size of the `destination` buffer might not be sufficient to accommodate the characters from the `source` buffer and terminate the buffer with a null. Polyspace reports a violation of the rule.

```
#include <string.h>

int main(int argc, char *argv[]) {
  const char *const source = (argc && argv[0]) ? argv[0] : "";
  char destination[128];
  strcpy(const_cast<char*>(source), destination);//Noncompliant

  return 0;
}
```

#### Correction — Allocate Sufficient Memory for Destination

This violation is resolved by allocating sufficient memory for the `destination` buffer. For instance, use the function `strlen()` to calculate the size of the `source` buffer and allocate sufficient memory for the `destination` buffer so that it can accommodate all characters from the `source` buffer and the null terminator (`'\0'` ).

```
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
  const char *const source = (argc && argv[0]) ? argv[0] : "";
  char* destination = (char *)malloc(strlen(source)+ 1);
  if(destination!=NULL){
      strcpy(const_cast<char*>(source), destination);//Compliant
  }else{
      /*Handle Error*/
  }
  //...
  free(destination);
  return 0;
}
```

## Check Information

**Group:** Input/output library
**Category:** Advisory, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A27-0-3

Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call

## Description

### Rule Definition

*Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call.*

### Rationale

Alternating input and output operations on a stream without an intervening flush or positioning call is undefined behavior.

### Polyspace Implementation

The checker checks for situations when:

- You do not perform a flush or function positioning call between an output operation and a following input operation on a file stream in update mode.

  To resolve the rule violation, call `fflush()` or a file positioning function such as `fseek()` or `fsetpos()` between output and input operations on an update stream.
- You do not perform a function positioning call between an input operation and a following output operation on a file stream in update mode.

  To resolve the rule violation, call a file positioning function between input and output operations on an update stream.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Read After Write Without Intervening Flush

```
#include <stdio.h>
#define SIZE20 20

void initialize_data(char* data, size_t s) {};
const char *temp_filename = "/tmp/demo.txt";

void funcNonCompliant()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;
    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Read operation after write without
    intervening flush. */
    if (fread(data, 1, SIZE20, file) < SIZE20)  //Noncompliant
      {
          (void)fclose(file);
          /* Handle error. */;
      }

    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}


void funcCompliant()
{
    char data[SIZE20];
    char append_data[SIZE20];
    FILE *file;

    file = fopen(temp_filename, "a+");
    if (file == NULL)
      {
        /* Handle error. */;
      }

    initialize_data(append_data, SIZE20);

    if (fwrite(append_data, 1, SIZE20, file) != SIZE20)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    /* Buffer flush after write and before read */
    if (fflush(file) != 0)
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    if (fread(data, 1, SIZE20, file) < SIZE20) //Compliant
      {
        (void)fclose(file);
        /* Handle error. */;
      }
    if (fclose(file) == EOF)
      {
        /* Handle error. */;
      }
}
```

In this example, in the function `funcNonCompliant`, the file `demo.txt` is opened for reading and appending. After the call to `fwrite()`, a call to `fread()` without an intervening flush operation is undefined behavior and violates the rule.

The function `funcCompliant` shows an alternative solution where a flush call is performed after writing data to the file and before calling `fread()`.

## Check Information

**Group:** Input/output library
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule A27-0-4

C-style strings shall not be used

## Description

### Rule Definition

*C-style strings shall not be used.*

### Rationale

The underlying character array that stores a C-style string has many disadvantages such as:

- You must explicitly handle memory allocation and deallocation if you perform operations on the string that require non-trivial memory manipulations.
- It is not always clear whether a `char*` points to a single character or to a C-style string.
- You might accidentally convert an array to a raw pointer when you pass it by value or by pointer to a function, which results in a loss of information about the array size (array decay). For example, in this code snippet, `func` prints the size of the pointer to the first character of `cString` (8) , while the actual size of `cString` is 6.

```
void func(char *c){ //function takes array by value
  cout << sizeof(c);
}

void main(){
  char cString[]{ "pizza" }; //Size is 6 (5 characters + null terminator)
  func(cString); // Size is 8 (size of char*)
}
```

Instead, use the `std::string` class to store a sequence of characters. The class handles allocations and deallocations, and instantiates an object that you can safely pass to functions. The class also has built-in functionalities to manipulate the string such as iterators.

### Polyspace Implementation

Polyspace flags the use of:

- Pointers to char (`char*`) and arrays of char (`char someArray[]`).
- Pointers to and arrays of char with a type qualifier such as `volatile` or `const`. For example `char const*`.
- Pointers to and arrays of type `wchar_t`, `char16_t`, and `char32_t`.

If you have a function declaration and its definition in your source code, Polyspace places the violation on the function definition. For example:

```
const char* greeter(void);
//....
const char* greeter(void){ //Non-compliant
  return "Hello";
}
```

Polyspace does not flag the use of:

- Pointers to or arrays of `signed` or `unsigned` char. For example, `signed_c` and `unsigned_arr` are not flagged in this code snippet:

  ```
  signed char* signed_c;
  unsigned char unsigned_arr[2048];
  ```
- Literal strings. For example, the return value of `greeter()` is not flagged in this code snippet, but the use of `const char*` in the first line is flagged:

  ```
  const char* greeter(void){ //Non-compliant
    return "Hello"; // Compliant
  }
  ```
- The parameters of `main()`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of C-Style Strings

```
#include<iostream>
#include <string>
#include<cstring>

char* sub_c_str(   //Non-compliant
    const char* str1, const int delim)  //Non-compliant
{
    size_t index {strlen(str1)};
    if (strchr(str1, delim)) {
        index = (size_t)(strchr(str1, delim) - str1);
    }
    char* p = (char*) malloc(index + 1); //Non-compliant
    //memory leak if p is not freed by caller
    strncpy(p, str1, index);
    return p;

}

std::string sub_str(std::string const str2, const char delim)
{
    return str2.substr(0, str2.find(delim));
}

int main()
{
    const char str1[] { "rootFolder/subFolder"}; // Non-compliant
    std::cout << sub_c_str(str1, '/') << std::endl;

    std::string const str2 { "rootFolder/subFolder" };
    std::cout << sub_str(str2, '/') << std::endl;
    return 0;
}
```

In this example, function `sub_c_str` returns a substring of C-style string parameter `str1` up to but not including the first instance of `delim`. The return type and first parameter of `sub_c_str` are both non-compliant pointers to char. Pointer `p`, which stores the substring, is also non-compliant. Note that if you do not free the memory allocated to `p` before the end of the program, this results in a memory leak.

Function `sub_str` takes advantage of the `std::string` class to perform the same operation as `sub_c_str`. The class handles memory allocation and deallocation. The class also has built-in functionalities (`find` and `subst`) to perform the string manipulation.

## Check Information

**Group:** Input/output library
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M0-1-1

A project shall not contain unreachable code

## Description

### Rule Definition

*A project shall not contain unreachable code.*

### Rationale

This rule flags situations where a group of statements is unreachable because of syntactic reasons. For instance, code following a `return` statement are always unreachable.

Unreachable code involve unnecessary maintenance and can often indicate programming errors.

### Polyspace Implementation

Polyspace reports a defect if a statement in your code is not reachable.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unreachable statements

```
int func(int arg) {
 int temp = 0;
 switch(arg) {
     temp = arg; // Noncompliant
     case 1:
     {
         break;
     }
     default:
     {
         break;
     }
 }
 return arg;
 arg++; // Noncompliant
}
```

These statements are unreachable:

- Statements inside a `switch` statement that do not belong to a `case` or `default` block.
- Statements after a `return` statement.

## Check Information

**Group:** Language Independent Issues
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M0-1-2

A project shall not contain infeasible paths

## Description

### Rule Definition

*A project shall not contain infeasible paths.*

### Rationale

This rule flags situations where a group of statements is redundant because of nonsyntactic reasons. For instance, an `if` condition is always true or false. Code that is unreachable from syntactic reasons are flagged by rule 0-1-1.

Unreachable or redundant code involve unnecessary maintenance and can often indicate programming errors.

### Polyspace Implementation

Polyspace Bug Finder reports a violation of this if your code contains dead code and unnecessary `if` conditionals. See `Dead code` and `Useless if` checkers.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Boolean Operations with Invariant Results

```
void func (unsigned int arg) {
 if (arg >= 0U) //Noncompliant
     arg  = 1U;
 if (arg < 0U) //Noncompliant
     arg = 1U;
}
```

An `unsigned int` variable is nonnegative. Both `if` conditions involving the variable are always true or always false and are therefore redundant.

## Check Information

**Group:** Language Independent Issues
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M0-1-3

A project shall not contain unused variables

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*A project shall not contain unused variables.*

### Rationale

Presence of unused variables indicates that the wrong variable name might be used in the source code. Removing these variables reduces the possibility of the wrong variable being used in further development. Keep padding bits in bitfields unnamed to reduce unused variables in your project.

### Polyspace Implementation

The checker flags local or global variables that are declared or defined but not read or written in any source files of the project. This specification also applies to members of structures and classes.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use of Named Bit Field for Padding

```
#include <iostream>
struct S {
    unsigned char b1 : 3;
    unsigned char pad: 1;  //Noncompliant
    unsigned char b2 : 4;
};
void init(struct S S_obj)
{
    S_obj.b1 = 0;
    S_obj.b2 = 0;
}
```

In this example, the bit field `pad` is used for padding the structure. Therefore, the field is never read or written and causes a violation of this rule. To avoid the violation, use an unnamed field for padding.

```
#include <iostream>
struct S {
    unsigned char b1 : 3;
    unsigned char : 1;  //Compliant
    unsigned char b2 : 4;
};
void init(struct S S_obj)
{
    S_obj.b1 = 0;
    S_obj.b2 = 0;
}
```

## Check Information

**Group:** Language Independent Issues
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M0-1-4

A project shall not contain non-volatile POD variables having only one use

## Description

### Rule Definition

*A project shall not contain non-volatile POD variables having only one use.*

### Rationale

If you use a non-volatile variable with a Plain Old Data type (`int`, `double`, etc.) *only once*, you can replace the variable with a constant literal. Your use of a variable indicates that you intended more than one use for that variable and might have a programming error in the code. You might have omitted the other uses of the non-volatile variable or incorrectly used other variables at intended points of use.

### Polyspace Implementation

The checker flags local and static variables that have a function scope (locally static) and file scope, which are used only once. The checker considers `const`-qualified global variables without the `extern` specifier as static variables with file scope.

The checker counts these use cases as one use of the non-volatile variable:

- An explicit initialization using a constant literal or the return value of a function
- An assignment
- A reference to the variable such as a read operation
- An assignment of the variable address to a pointer

If the variable address is assigned to a pointer, the checker assumes that the pointer might be dereferenced later and does not flag the variable.

Some objects are designed to be used only once by their semantics. Polyspace does not flag a single use of these objects:

- `lock_guard`
- `scoped_lock`
- `shared_lock`
- `unique_lock`
- `thread`
- `future`
- `shared_future`

If you use nonstandard objects that provide similar functionality as the objects in the preceding list, Polyspace might flag single uses of the nonstandard objects. Justify their single uses by using comments.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Non-volatile Variable Used Only Once

```
#include <mutex>
int readStatus1();
int readStatus2();
void getReading(int*);
extern std::mutex m;
void foo()
{
    // Initiating lock 'lk'
    std::lock_guard<std::mutex> lk{m};
    int checkEngineStatus1 = readStatus1();
    int checkEngineStatus2 = readStatus2();//Noncompliant
    int sensorData;//Noncompliant
    getReading(&sensorData);
    if(checkEngineStatus1) {
        //Perform some actions if both statuses are valid
    }
    // Release lock when 'lk' is deleted at exit point of scope
}
```

In this example, the variable `checkEngineStatus2` is used only once. The single use of this variable might indicate a programming error. For instance, you might have intended to check both `checkEngineStatus1` and `checkEngineStatus2` in the `if` condition, but omitted the second check. The variable `sensorData` is also used only once when its address is passed to the function `getReading()`. Polyspace flags these single use variables.

The `lock_guard` object `lk` is used a single time. Because the semantics of a `lock_guard` object justifies its single use, Polyspace does not flag it.

## Check Information

**Group:** Language independent issues
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M0-1-8

All functions with void return type shall have external side effect(s)

## Description

### Rule Definition

*All functions with void return type shall have external side effect(s).*

### Rationale

A function that has a `void` return type does not return anything. When such a function has no external side effects, it does not contribute to the output and consumes time. That these functions have no contribution to the output of the program might be contrary to developer expectation. Such a function might be unnecessary or indicate issues in the program design. Avoid such functions.

The MISRA C++:2008 standard considers these effects as external side effects:

- Reading or writing in resources such as a file or a stream.
- Changing the value of a nonlocal variable.
- Changing the value of a reference type argument.
- Using a `volatile` object.
- Raising an exception.

### Polyspace Implementation

Polyspace flags the definition of a `void` type function if the function has no side effects.

Polyspace considers a function to have side effects if the function performs any of these tasks:

- Calls an impure function other than itself.
- Changes the value or dereferences a reference or pointer type argument.
- Changes the value or deferences of a nonlocal variable.
- Contains assembly instructions.
- Accesses the `this` pointer of its parent class.
- Raises an exception. Raising exceptions is considered as a side effect even if the function does not exit with an exception.
- Writes into an absolute address by using a local pointer.
- Accesses a `volatile` object, class member, or `struct` member.
- Sets the value of a class or `struct` member.

This checker does not flag standard implementations of placement `new` and `delete` functions.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid void Functions That Have No Side Effects

```
#include<cmath>
volatile int flag;
struct reg2 {
    struct reg1 {
      unsigned int U;
    } REG1;
    unsigned int* P;
  };

void init ( int refToInt )          // Noncompliant
{
    refToInt = 0;
}

void transform(double theta) //Noncompliant
{
    theta = sin(theta)/(1-cos(theta));
}
void foo(int val){ //Noncompliant
    if(val){
        foo(val--);
    }
}

void rd(void){ //Compliant
    if(flag==0){
        //...
    }
}
void read_reg(void){ //Compliant
     reg2* reg = reinterpret_cast<reg2 *>((0x02008000));
     reg->REG1.U = 0;
     //...
}
```

In this example, Polyspace flags the `void` functions that have no side effects. For instance:

- The function `init()` initializes a local variable. This function returns no value and has no additional side effects. Perhaps the function is intended to initialize a reference and you missed an & in the declaration. Polyspace flags the function.
- The function `transform()` transforms a number to another number by calling the pure functions `sin()` and `cos()`. The function returns nothing and has no side effects. Perhaps the function is intended to return the transformed number and you missed a return statement. Polyspace flags the function.
- The function `foo()` sets up a recursion but has no effect on the output. Perhaps the recursion is set up improperly. Polyspace flags the function.
- The function `rd()` reads a `volatile` object `flag`. Polyspace does not raise a violation on this function.
- The function `read_reg()` writes into the absolute address `0x02008000` by using the local pointer `reg`. Polyspace assumes that the function might have a side effects and does not raise a violation on it.

## Check Information

**Group:** Language independent issues
**Category:** Required, Automated

## Version History

**Introduced in R2022a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M0-1-9

There shall be no dead code

## Description

### Rule Definition

*There shall be no dead code.*

### Rationale

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code. For instance, suppose that a variable is never read following a write operation. The write operation is redundant.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Redundant Operations

```
#define ULIM 10000

int func(int arg) {
    int res;
    res = arg*arg + arg;
    if (res > ULIM)
        res = 0; //Noncompliant
    return arg;
}
```

In this example, the operations involving `res` are redundant because the function `func` returns its argument `arg`. All operations involving `res` can be removed without changing the effect of the function.

The checker flags the last write operation on `res` because the variable is never read after that point. The dead code can indicate an unintended coding error. For instance, you intended to return the value of `res` instead of `arg`.

## Check Information

**Group:** Language Independent Issues
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M0-1-10

Every defined function should be called at least once

## Description

### Rule Definition

*Every defined function should be called at least once.*

### Rationale

If a function with a definition is not called, it might indicate a serious coding error. For instance, the function call is unreachable or a different function is called unintentionally.

### Polyspace Implementation

The checker detects situations where a static function is defined but not called at all in its translation unit.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Uncalled Static Function

```
static void func1() {
}

static void func2() { //Noncompliant
}

void func3();

int main() {
    func1();
    return 0;
}
```

The `static` function `func2` is defined but not called.

The function `func3` is not called either, however, it is only declared and not defined. The absence of a call to `func3` does not violate the rule.

## Check Information

**Group:** Language Independent Issues
**Category:** Advisory, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M0-2-1

An object shall not be assigned to an overlapping object

## Description

### Rule Definition

*An object shall not be assigned to an overlapping object.*

### Rationale

When you assign an object to another object with overlapping memory, the behavior is undefined.

The exceptions are:

- You assign an object to another object with exactly overlapping memory and compatible type.
- You copy one object to another with `memmove`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Assignment of Union Members

```
void func (void) {
    union {
        short i;
        int j;
    } a = {0}, b = {1};

    a.j = a.i;   //Noncompliant
    a = b;       //Compliant
}
```

In this example, the rule is violated when `a.i` is assigned to `a.j` because the two variables have overlapping regions of memory.

## Check Information

**Group:** Language Independent Issues
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**
"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M0-3-2

If a function generates error information, then that error information shall be tested

## Description

### Rule Definition

*If a function generates error information, then that error information shall be tested.*

### Rationale

If you do not check the return value of functions that indicate error information through their return values, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

For the `errno`-setting functions, to see if the function call completed without errors, check `errno` for error values. The return values of these `errno`-setting functions do not indicate errors. The return value can be one of the following:

- `void`
- Even if an error occurs, the return value can be the same as the value from a successful call. Such return values are called in-band error indicators. For instance, `strtol` converts a string to a long integer and returns the integer. If the result of conversion overflows, the function returns `LONG_MAX` and sets `errno` to `ERANGE`. However, the function can also return `LONG_MAX` from a successful conversion. Only by checking `errno` can you distinguish between an error and a successful conversion.

For the `errno`-setting functions, you can determine if an error occurred only by checking `errno`.

### Polyspace Implementation

The checker raises a violation when:

- You call sensitive functions that return information about possible errors and then you ignore the return value or use the output of the function without testing the return value.

  The checker covers function from the standard library and other well-known libraries such as the POSIX library or the WinAPI library. Polyspace considers a function as sensitive if the function call is prone to failure because of reasons such as:

  - Exhausted system resources (for example, when allocating resources).
  - Changed privileges or permissions.
  - Tainted sources when reading, writing, or converting data from external sources.
  - Unsupported features despite an existing API.

  Polyspace considers a function a critical sensitive when they perform critical tasks such as:

  - Set privileges (for example, `setuid`)
  - Create a jail (for example, `chroot`)

  - Create a process (for example, `fork`)
  - Create a thread (for example, `pthread_create`)
  - Lock or unlock mutex (for example, `pthread_mutex_lock`)
  - Lock or unlock memory segments (for example, `mlock`)

  For functions that are not critical, the checker is not flagged if you explicitly ignore the return value by casting it to `void`. Explicitly ignoring the return value of critical sensitive functions is flagged by Polyspace.

- You call a function that sets `errno` to indicate error conditions, but do not check `errno` after the call. For these functions, checking `errno` is the only reliable way to determine if an error occurred.

  Functions that set `errno` on errors include:

  - `fgetwc`, `strtol`, and `wcstol`.

    For a comprehensive list of functions, see documentation about errno.
  - POSIX `errno`-setting functions such as `encrypt` and `setkey`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Sensitive Function Return Ignored

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>
#include <cstdlib>
#define fatal_error() abort()

void initialize_1() {
    pthread_attr_t attr;
    pthread_attr_init(&attr); //Noncompliant
}

void initialize_2() {
    pthread_attr_t attr;
   (void)pthread_attr_init(&attr); //Compliant
}

void initialize_3() {
    pthread_attr_t attr;
    int result;
    result = pthread_attr_init(&attr); //Compliant
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

```
int read_file_1(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r");
  return 0; //Noncompliant

}
int read_file_2(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r"); //Compliant
  if (in==NULL){
      // Handle error
  }
  return 0;
}
```

This example shows a call to the sensitive functions `pthread_attr_init` and `fmemopen`. Polyspace raises a flag if:

- You implicitly ignore the return of the sensitive function. Explicitly ignoring the output of sensitive functions is not flagged.
- You obtain the return value of a sensitive function but do not test the value before exiting the relevant scope. The violation is raised on the exit statement.

To be compliant, you can explicitly cast their return value to `void` or test the return values to check for errors.

### Critical Function Return Ignored

```
#include <pthread.h>
#include <cstdlib>
#define fatal_error() abort()
extern void *start_routine(void *);

void returnnotchecked_1() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0)); //Noncompliant
    pthread_join(thread_id,  &res); //Noncompliant
}

void returnnotchecked_2() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL); //Compliant
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res); //Compliant
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), the rule checker still raises a violation. The other critical function, `pthread_join`, returns a value that is ignored implicitly.

To be compliant, check the return value of these critical functions to verify the function performed as expected.

### errno Not Checked After Call to strtol

```
#include<cstdlib>
#include<cerrno>
#include<climits>
#include<iostream>

int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    long val = strtol(str, &endptr, base); //Noncompliant
    std::cout<<"Return value of strtol() = %ld\n" << val;

    errno = 0;
    long val2 = strtol(str, &endptr, base); //Compliant
    if((val2 == LONG_MIN || val2 == LONG_MAX) && errno == ERANGE) {
         std::cout<<"strtol error";
         exit(EXIT_FAILURE);
    }
    std::cout<<"Return value of strtol() = %ld\n" << val2;
}
```

In the noncompliant example, the return value of `strtol` is used without checking `errno`.

To be compliant, before calling `strtol`, set `errno` to zero . After a call to `strtol`, check the return value for `LONG_MIN` or `LONG_MAX` and `errno` for `ERANGE`.

## Check Information

**Group:** Language independent issues
**Category:** Required, Non-automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M2-7-1

The character sequence /* shall not be used within a C-style comment

## Description

### Rule Definition

*The character sequence /* shall not be used within a C-style comment.*

### Rationale

If your code contains a `/*` in a `/* */` comment, it typically means that you have inadvertently commented out code. See the example that follows.

### Polyspace Implementation

You cannot justify a violation of this rule using source code annotations.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use of `/*` in `/* */` Comment

```
void setup(void);
void foo() {
    /* Initializer functions
     setup();
    /* Step functions */  //Noncompliant
}
```

In this example, the call to `setup()` is commented out because the ending `*/` is omitted, perhaps inadvertently. The checker flags this issue by highlighting the `/*` in the `/* */` comment.

## Check Information

**Group:** Lexical Conventions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M2-10-1

Different identifiers shall be typographically unambiguous

## Description

### Rule Definition

*Different identifiers shall be typographically unambiguous.*

### Rationale

When you use different identifiers that look typographically similar, you might inadvertently use the incorrect identifier later in your code. Such typographically ambiguous identifiers might lead to bugs that are difficult to diagnose.

Use identifiers that are unambiguously distinct. Avoid using identifiers that differ by a combination of these:

- The use of a lowercase letter instead of an uppercase one, and vice versa.
- The presence or absence of the underscore character.
- The interchange of the letter `O` and the digit `0`.
- The interchange of the letter `I` and the digit `1`.
- The interchange of the letter `I` and the letter `l`.
- The interchange of the letter `S` and the digit `5`.
- The interchange of the letter `Z` and the digit `2`.
- The interchange of the letter `n` and the letter `h`.
- The interchange of the letter `B` and the digit `8`.
- The interchange of the letters `rn` and the letter `m`.

Depending on the font you use, the preceding characters might look similar.

### Polyspace Implementation

Polyspace reports a defect if two identifiers differ by a combination of ambiguous characters mentioned in the preceding list. The rule checker does not consider the fully qualified names of variables when checking this rule.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val;  //Non-compliant

    int id2_numval;
    int id2_numVal;   //Non-compliant

    int id3_lvalue;
    int id3_Ivalue;   //Non-compliant

    int id4_xyZ;
    int id4_xy2;      //Non-compliant

    int id5_zerO;
    int id5_zer0;     //Non-compliant

    int id6_rn;
    int id6_m;        //Non-compliant
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

## Check Information

**Group:** Lexical Conventions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M2-13-2

Octal constants (other than zero) and octal escape sequences (other than "\0" ) shall not be used

## Description

### Rule Definition

*Octal constants (other than zero) and octal escape sequences (other than "\0" ) shall not be used.*

### Rationale

Octal constants are denoted by a leading zero. A developer or code reviewer can mistake an octal constant as a decimal constant with a redundant leading zero.

Octal escape sequences beginning with \ can also cause confusion. Inadvertently introducing an 8 or 9 in the digit sequence after \ breaks the escape sequence and introduces a new digit. A developer or code reviewer can ignore this issue and continue to treat the escape sequence as one digit.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Octal Constants and Octal Escape Sequences

```
void func(void) {
  int busData[6];

  busData[0] = 100;
  busData[1] = 108;
  busData[2] = 052;      //Noncompliant
  busData[3] = 071;      //Noncompliant
  busData[4] = '\109';   //Noncompliant
  busData[5] = '\100';   //Noncompliant

}
```

The checker flags all octal constants (other than zero) and all octal escape sequences (other than `\0`).

In this example:

- The octal escape sequence contains the digit 9, which is not an octal digit. This escape sequence has implementation-defined behavior.
- The octal escape sequence `\100` represents the number 64, but the rule checker forbids this use.

## Check Information

**Group:** Lexical Conventions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M2-13-3

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type

## Description

### Rule Definition

*A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.*

### Rationale

The signedness of a constant is determined from:

- Value of the constant.
- Base of the constant: octal, decimal or hexadecimal.
- Size of the various types.
- Any suffixes used.

Unless you use a suffix `u` or `U`, another developer looking at your code cannot determine easily whether a constant is signed or unsigned.

### Polyspace Implementation

Polyspace reports a violation of this rule if an unsigned octal or hexadecimal constant does not have the suffix `U`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Lexical Conventions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M2-13-4

Literal suffixes shall be upper case

## Description

### Rule Definition

*Literal suffixes shall be upper case.*

### Rationale

Literal constants can end with the letter `l` (el). Enforcing literal suffixes to be upper case removes potential confusion between the letter `l` and the digit `1`.

For consistency, use upper case constants for other suffixes such as `U` (unsigned) and `F` (float).

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Literal Constants with Lower Case Suffix

```
const int a = 0l; //Noncompliant
const int b = 0L; //Compliant
```

In this example, both `a` and `b` are assigned the same literal constant. However, from a quick glance, one can mistakenly assume that `a` is assigned the value `01` (octal one).

## Check Information

**Group:** Lexical Conventions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M3-1-2

Functions shall not be declared at block scope

## Description

### Rule Definition

*Functions shall not be declared at block scope.*

### Rationale

It is a good practice to place all declarations at the namespace level.

Additionally, if you declare a function at block scope, it is often not clear if the statement is a function declaration or an object declaration with a call to the constructor.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Function Declarations at Block Scope

```
class A {
};

void b1() {
    void func(); //Noncompliant
    A a();   //Noncompliant
}
```

In this example, the declarations of `func` and `a` are in the block scope of `b1`.

The second function declaration can cause confusion because it is not clear if `a` is a function that returns an object of type `A` or `a` is itself an object of type `A`.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M3-2-1

All declarations of an object or function shall have compatible types

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”*.

### Rule Definition

*All declarations of an object or function shall have compatible types.*

### Rationale

If the declarations of an object or function in two different translation units have incompatible types, the behavior is undefined.

### Polyspace Implementation

Polyspace considers two types to be compatible if they have the same size and signedness in the environment that you use. The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` or `extern` functions
- Uncalled and undefined local functions
- Unused types and variables

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Compatible and Incompatible Definitions in Two Files

**file1.cpp**

```
typedef          char         char_t;
typedef signed   short        int16_t;
typedef signed   long         int64_t;

namespace bar {
    int64_t a;
    int16_t c;

};
```

**file2.cpp**

```
typedef          char         char_t;
typedef signed   int          int32_t;

namespace bar {
    extern char_t c;// Noncompliant
    extern int32_t a;
    void foo(void){
        ++a;
        ++c;
    }
};
```

In this example, the variable `bar::c` is defined as a `char` in `file2.cpp` and as a `signed short` in `file1.cpp`. In the target processor i386, the size of these types are not equal. Polyspace flags the definition of `bar::c`.

The variable `bar::a` is defined as a `long` in `file1.cpp` and as an `int` in `file2.cpp`. In the target processor i386, both `int` and `long` has a size of 32 bits. Because the definitions of `bar::a` is compatible in both files, Polyspace does not raise a flag.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M3-2-2

The One Definition Rule shall not be violated

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

### Rule Definition

*The One Definition Rule shall not be violated.*

### Rationale

Violations of the One Definition Rule leads to undefined behavior.

### Polyspace Implementation

The checker flags situations where the same function or object has multiple definitions and the definitions differ by some token. The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` or `extern` functions
- Uncalled and undefined local functions
- Unused types and variables

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Different Tokens in Same Type Definition

This example uses two files:

- `file1.cpp`:

  ```
  typedef struct S //Noncompliant
  {
     int x;
     int y;
  }S;
  void foo(S& s){
  //...
  }
  ```
- `file2.cpp`:

  ```
  typedef struct S
  ```

```
{
   int y;
   int x;
}S ;
void bar(S& s){
//...
}
```

In this example, both `file1.cpp` and `file2.cpp` define the structure `S`. However, the definitions switch the order of the structure fields.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M3-2-3

A type, object or function that is used in multiple translation units shall be declared in one and only one file

## Description

### Rule Definition

*A type, object or function that is used in multiple translation units shall be declared in one and only one file.*

### Rationale

If you declare an identifier in a header file, you can include the header file in any translation unit where the identifier is defined or used. In this way, you ensure consistency between:

- The declaration and the definition.
- The declarations in different translation units.

The rule enforces the practice of declaring external objects or functions in header files.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M3-2-4

An identifier with external linkage shall have exactly one definition

## Description

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

### Rule Definition

*An identifier with external linkage shall have exactly one definition.*

### Rationale

If an identifier has multiple definitions or no definitions, it can lead to undefined behavior.

### Polyspace Implementation

The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` functions or `extern` functions
- Uncalled and undefined local functions
- Unused types and variables

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Multiple Definitions of Identifier

This example uses two files:

- `file1.cpp`:

```
typedef signed   int          int32_t;

namespace NS {
    extern int32_t a;

    void foo(){
        a = 0;

    }
};
```
- `file2.cpp`:

```
typedef signed   int          int32_t;
typedef signed   long long         int64_t;

namespace NS {
    extern int64_t a; //Noncompliant
    void bar(){
        ++a;

    }
};
```

The same identifier `a` is defined in both files.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M3-3-2

If a function has internal linkage then all re-declarations shall include the static storage class specifier

## Description

### Rule Definition

*If a function has internal linkage then all re-declarations shall include the static storage class specifier.*

### Rationale

If a function declaration has the `static` storage class specifier, it has internal linkage. Subsequent redeclarations of the function have internal linkage even without the `static` specifier.

However, if you do not specify the `static` keyword explicitly, it is not immediately clear from a declaration whether the function has internal linkage.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Missing `static` Specifier from Redeclaration

```
static void func1 ();
static void func2 ();

void func1() {}  //Noncompliant
static void func2() {}
```

In this example, the function `func1` is declared `static` but defined without the `static` specifier.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M3-4-1

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility

## Description

### Rule Definition

*An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.*

### Rationale

Defining variables with the minimum possible block scope reduces the possibility that they might later be accessed unintentionally.

For instance, if an object is meant to be accessed in one function only, declare the object local to the function.

### Polyspace Implementation

The rule checker determines if an object is used in one block only. If the object is used in one block but defined outside the block, the checker raises a violation.

When you declare a variable outside a range-based `for` loop and use it only inside the loop block, Polyspace flags the variable. If you cannot declare the variable inside the loop block, justify this result using comments in your result or code. See “Address Results in Polyspace User Interface Through Bug Fixes or Justifications”.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Global Variable in Single Function

```
static int countReset; //Noncompliant

volatile int check;

void increaseCount() {
    int count = countReset;
    while(check%2) {
     count++;
    }
}
```

In this example, the variable `countReset` is declared global used in one function only. A compliant solution declares the variable local to the function to reduce its visibility.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M3-9-1

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations

## Description

### Rule Definition

*The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.*

### Rationale

If a redeclaration is not token-for-token identical to the previous declaration, it is not clear from visual inspection which object or function is being redeclared.

### Polyspace Implementation

The rule checker compares the current declaration with the last seen declaration.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Identical Declarations That Do Not Match Token for Token

```
typedef int* intptr;

int* map;
extern intptr map; //Noncompliant

intptr table;
extern intptr table; //Compliant
```

In this example, the variable `map` is declared twice. The second declaration uses a `typedef` which resolves to the type of the first declaration. Because of the `typedef`, the second declaration is not token-for-token identical to the first.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M3-9-3

The underlying bit representations of floating-point values shall not be used

## Description

### Rule Definition

*The underlying bit representations of floating-point values shall not be used.*

### Rationale

The underlying bit representations of floating point values vary across compilers. If you directly use the underlying representation of floating point values, your program is not portable across implementations.

### Polyspace Implementation

The rule checker flags conversions from pointers to floating point types into pointers to integer types, and vice versa.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Using Underlying Representation of Floating-Point Values

```
float fabs2(float f) {
    unsigned int* ptr = reinterpret_cast <unsigned int*> (&f); //Noncompliant
    *(ptr + 3) &= 0x7f;
    return f;
}
```

In this example, the `reinterpret_cast` attempts to cast a floating-point value to an integer and access the underlying bit representation of the floating point value.

## Check Information

**Group:** Basic Concepts
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M4-5-1

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and ! =, the unary & operator, and the conditional operator

## Description

### Rule Definition

*Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and ! =, the unary & operator, and the conditional operator.*

### Rationale

Operators other than the ones mentioned in the rule do not produce meaningful results with `bool` operands. Use of `bool` operands with these operators can indicate programming errors. For instance, you intended to use the logical operator `||` but used the bitwise operator `|` instead.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Compliant and Noncompliant Uses of `bool` Operands

```
void boolOperations() {
    bool lhs = true;
    bool rhs = false;

    int res;

    if(lhs & rhs) {} //Noncompliant
    if(lhs < rhs) {} //Noncompliant
    if(~rhs) {}      //Noncompliant
    if(lhs ^ rhs) {} //Noncompliant
    if(lhs == rhs) {} //Compliant
    if(!rhs) {}       //Compliant
    res = lhs? -1:1;  //Compliant
}
```

In this example, `bool` operands do not violate the rule when used with the `==`, `!` and the `?` operators.

## Check Information

**Group:** Standard Conversions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M4-5-3

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and ! =, and the unary & operator

## Description

### Rule Definition

*Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and ! =, and the unary & operator.*

### Rationale

The C++03 Standard only requires that the characters `'0'` to `'9'` have consecutive values. Other characters do not have well-defined values. If you use these characters in operations other than the ones mentioned in the rule, you implicitly use their underlying values and might see unexpected results.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Compliant and Noncompliant Uses of Character Operands

```
void charManipulations (char ch) {

    char initChar = 'a'; //Compliant
    char finalChar = 'z'; //Compliant

    if(ch == initChar) {} //Compliant
    if( (ch >= initChar) && (ch <= finalChar)) {} //Noncompliant
    else if( (ch >= '0') && (ch <= '9') ) {} //Compliant by exception
}
```

In this example, character operands do not violate the rule when used with the = and == operators. Character operands can also be used with relational operators as long as the comparison is performed with the digits `'0'` to `'9'`.

## Check Information

**Group:** Standard Conversions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M4-10-1

NULL shall not be used as an integer value

## Description

### Rule Definition

*NULL shall not be used as an integer value.*

### Rationale

In C++, you can use the literals 0 and NULL as both an integer and a null pointer constant. However, use of 0 as a null pointer constant or NULL as an integer can cause developer confusion.

This rule restricts the use of NULL to null pointer constants. `AUTOSAR C++14 Rule M4-10-2` restricts the use of the literal 0 to integers.

### Polyspace Implementation

The checker flags assignment of NULL to an integer variable or binary operations involving NULL and an integer. Assignments can be direct or indirect such as passing NULL as integer argument to a function.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Compliant and Noncompliant Uses of `NULL`

```
#include <cstddef>

void checkInteger(int);
void checkPointer(int *);

void main() {
    checkInteger(NULL); //Noncompliant
    checkPointer(NULL); //Compliant
}
```

In this example, the use of `NULL` as argument to the `checkInteger` function is noncompliant because the function expects an `int` argument.

## Check Information

**Group:** Standard Conversions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M4-10-2

Literal zero (0) shall not be used as the null-pointer-constant

## Description

### Rule Definition

*Literal zero (0) shall not be used as the null-pointer-constant.*

### Rationale

In C++, you can use the literals 0 and NULL as both an integer and a null pointer constant. However, use of 0 as a null pointer constant or NULL as an integer can cause developer confusion.

This rule restricts the use of the literal 0 to integers. `AUTOSAR C++14 Rule M4-10-1` restricts the use of NULL to null pointer constants.

### Polyspace Implementation

The checker flags assignment of 0 to a pointer variable or binary operations involving 0 and a pointer. Assignments can be direct or indirect such as passing 0 as pointer argument to a function.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Compliant and Noncompliant Uses of Literal 0

```
#include <cstddef>

void checkInteger(int);
void checkPointer(int *);

void main() {
    checkInteger(0); //Compliant
    checkPointer(0); //Noncompliant
}
```

In this example, the use of 0 as argument to the `checkPointer` function is noncompliant because the function expects an `int *` argument.

## Check Information

**Group:** Standard Conversions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-0-2

Limited dependence should be placed on C++ operator precedence rules in expressions

## Description

### Rule Definition

*Limited dependence should be placed on C++ operator precedence rules in expressions.*

### Rationale

Use parentheses to clearly indicate the order of evaluation.

Depending on operator precedence can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:
  - In the operation `*p++`, it is possible that you expect the dereferenced value to be incremented. However, the pointer `p` is incremented before the dereference.
  - In the operation `(x == y | z)`, it is possible that you expect `x` to be compared with `y | z`. However, the `==` operation happens before the `|` operation.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Evaluation Order Dependent on Operator Precedence Rules

```
#include <cstdio>

void showbits(unsigned int x) {
    for(int i = (sizeof(int) * 8) - 1; i >= 0; i--) {
       (x & 1u << i) ? putchar('1') : putchar('0'); // Noncompliant
    }
    printf("\n");
}
```

In this example, the checker flags the operation `x & 1u << i` because the statement relies on operator precedence rules for the << operation to happen before the & operation. If this is the intended order, the operation can be rewritten as `x & (1u << i)`.

## Check Information

**Group:** Expressions
**Category:** Advisory, Partially automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-0-3

A cvalue expression shall not be implicitly converted to a different underlying type

## Description

### Rule Definition

*A cvalue expression shall not be implicitly converted to a different underlying type.*

### Rationale

This rule ensures that the result of the expression does not overflow when converted to a different type.

### Polyspace Implementation

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

The underlying data type of a cvalue expression is the widest of operand data types in the expression. For instance, if you add two variables, one of type `int8_t` (`typedef` for `char`) and another of type `int32_t` (`typedef` for `int`), the addition has underlying type `int32_t`. If you assign the sum to a variable of type `int8_t`, the rule is violated.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Implicit Conversion of Cvalue Expression

```
#include<cstdint>

void func ( )
  {
    int32_t s32;
    int8_t s8;
    s32 = s8 + s8; //Noncompliant
    s32 = s32 + s8; //Compliant
  }
```

In this example, the rule is violated when two variables of type `int8_t` are added and the result is assigned to a variable of type `int32_t`. The underlying type of the addition does not take into account the integer promotion involved and is simply the widest of operand data types, in this case, `int8_t`.

The rule is not violated if one of the operands has type `int32_t` and the result is assigned to a variable of type `int32_t`. In this case, the underlying data type of the addition is the same as the type of the variable to which the result is assigned.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-0-4

An implicit integral conversion shall not change the signedness of the underlying type

## Description

### Rule Definition

*An implicit integral conversion shall not change the signedness of the underlying type.*

### Rationale

Some conversions from signed to unsigned data types can lead to implementation-defined behavior. You can see unexpected results from the conversion.

### Polyspace Implementation

The checker flags implicit conversions from a signed to an unsigned integer data type or vice versa.

The checker assumes that `ptrdiff_t` is a signed integer.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Implicit Conversions that Change Signedness

```
typedef char int8_t;
typedef unsigned char uint8_t;

void func()
  {
    int8_t s8;
    uint8_t u8;

    s8 = u8; //Noncompliant
    u8 = s8 + u8; //Noncompliant
    u8 = static_cast< uint8_t > ( s8 ) + u8; //Compliant
}
```

In this example, the rule is violated when a variable with a variable with signed data type is implicitly converted to a variable with unsigned data type or vice versa. If the conversion is explicit, as in the preceding example, the rule violation does not occur.

## Check Information

**Group:** Expressions<br>
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-0-5

There shall be no implicit floating-integral conversions

## Description

### Rule Definition

*There shall be no implicit floating-integral conversions.*

### Rationale

If you convert from a floating point to an integer type, you lose information. Unless you explicitly cast from floating point to an integer type, it is not clear whether the loss of information is intended. Additionally, if the floating-point value cannot be represented in the integer type, the behavior is undefined.

Conversion from an integer to floating-point type can result in an inexact representation of the value. The error from conversion can accumulate over later operations and lead to unexpected results.

### Polyspace Implementation

The checker flags implicit conversions between floating-point types (`float` and `double`) and integer types (`short`, `int`, etc.).

This rule takes precedence over M5-0-4 and M5-0-6 if they apply at the same time.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Conversion Between Floating Point and Integer Types

```
typedef signed int int32_t;
typedef float float32_t;

void func ( )
  {
    float32_t f32;
    int32_t   s32;
    s32 = f32;   //Noncompliant
    f32 = s32;   //Noncompliant
    f32 = static_cast< float32_t > ( s32 ); //Compliant
  }
```

In this example, the rule is violated when a floating-point type is *implicitly* converted to an integer type. The violation does not occur if the conversion is explicit.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-0-6

An implicit integral or floating-point conversion shall not reduce the size of the underlying type

## Description

### Rule Definition

*An implicit integral or floating-point conversion shall not reduce the size of the underlying type.*

### Rationale

A conversion that reduces the size of the underlying type can result in loss of information.

### Polyspace Implementation

If the conversion is to a narrower integer with a different sign, then rule M5-0-4 takes precedence over rule M5-0-6. Only rule M5-0-4 is shown.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-0-7

There shall be no explicit floating-integral conversions of a cvalue expression

## Description

### Rule Definition

*There shall be no explicit floating-integral conversions of a cvalue expression.*

### Rationale

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation. For instance, in this example, the result of an integer division is then cast to a floating-point type.

```
short num;
short den;
float res;
res= static_cast<float> (num/den);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a floating-point division because of the later cast.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Conversion of Division Result from Integer to Floating Point

```
void func() {
     short num;
     short den;
     short res_short;
     float res_float;

     res_float = static_cast<float> (num/den); //Noncompliant

     res_short = num/den;
     res_float = static_cast<float> (res_short); //Compliant

}
```

In this example, the first cast on the division result violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the expression is evaluated with an underlying type `float`.
- The second cast makes it clear that the expression is evaluated with the underlying type `short`. The result is then cast to the type `float`.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-0-8

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression

## Description

### Rule Definition

*An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.*

### Rationale

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation. For instance, in this example, the sum of two `short` operands is cast to the wider type `int`.

```
short op1;
short op2;
int res;
res= static_cast<int> (op1 + op2);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a sum with the underlying type `int` because of the later cast.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Conversion of Sum to Wider Integer Type

```
void func() {
     short op1;
     short op2;
     int res;

     res = static_cast<int> (op1 + op2); //Noncompliant
     res = static_cast<int> (op1) + op2; //Compliant

}
```

In this example, the first cast on the sum violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the sum is evaluated with an underlying type `int`.
- The second cast first converts one of the operands to `int` so that the sum is actually evaluated with the underlying type `int`.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-0-9

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression

## Description

### Rule Definition

*An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.*

### Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression).. For instance, in this example, the sum of two `unsigned int` operands is cast to the type `int`.

```
unsigned int op1;
unsigned int op2;
int res;
res= static_cast<int> (op1 + op2);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a sum with the underlying type `int` because of the later cast.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Conversion of Sum to Wider Integer Type

```
typedef int int32_t;
typedef unsigned int uint32_t;

void func() {
     uint32_t op1;
     uint32_t op2;
     int32_t res;

     res = static_cast<int32_t> (op1 + op2); //Noncompliant
     res = static_cast<int32_t> (op1) +
           static_cast<int32_t> (op2); //Compliant

}
```

In this example, the first cast on the sum violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the sum is evaluated with an underlying type `int32_t`.
- The second cast first converts each of the operands to `int32_t` so that the sum is actually evaluated with the underlying type `int32_t`.

### Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-0-10

If the bitwise operators ~and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand

## Description

### Rule Definition

*If the bitwise operators ~and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.*

### Rationale

When the bitwise operators ~ and << are applied to small integer types, such as unsigned short and unsigned char, the operations are preceded by integral promotion. That is, the small integer types are first promoted to a larger integer type, and then the operation takes place. The result of these bitwise operation might contain unexpected higher order bits. For instance:

```
uint8_t var = 0x5aU;
uint8_t result = (~var)>>4;
```

The binary representation of `var` is `0101 1010` and that of `~var` is `1010 0101`. You might expect that `result` is `0000 1010`. Because `var` is promoted to a larger integer before `~var` is calculated, result becomes `1111 1010`. The higher order bits might be unexpected. The results of such operations might depend on the size of `int` in your implementation.

To avoid confusion and unexpected errors, cast the result of the bitwise ~ and >> operators back to the underlying type of the operands before using the results. For instance:

```
uint8_t var = 0x5aU;
uint8_t result = (static_cat<unit8_t>(~var))>>4;
```

The binary representation of `result` in this case is `0000 1010`, which is the expected value.

As an exception, casting is not required if you apply these bitwise operators on short integer types, and then immediately assign the result to an object of the same underlying type. For instance, the value of result in this case is `0000 1010` without requiring a cast.

```
uint8_t var = 0x5aU;
unit8_t result = ~var; // No higher order bits
                       // due to implicit conversion
uint8_t result = results>>4;
```

### Polyspace Implementation

Polyspace flags the use of the bitwise ~ and >> operators if all of these conditions are true:

- The operators are used on an `unsigned short` or `unsigned char` operand.
- The result of the operation is not immediately assigned to an object that has the same underlying type as the operand.
- The result is used without being cast to the underlying type of the operand.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Cast Results of ~ and << Operators to the Operand Type When the Operand Is Small Integer Type

```
#include<cstdint>
void foo(){
    uint8_t var = 0x5aU;
    uint8_t result;
    result = ( ~var ) >> 4; // Non-compliant
    result = static_cast<uint8_t>(( ~var )) >> 4; // Compliant
    uint8_t cbe = ~var;//Compliant by Exception
}
```

In this example, Polyspace flags the use of ~ on the small integer `var`. The ~ operator is flagged because:

- It operates on an unsigned short integer `var`.
- The result of the operator is used in an expression without casting ~`var` to `uint8_t`.

When the result of ~ operator is cast to `unit8_t`, the use is compliant with this rule. When the result of ~ is immediately assigned to a `unit8_t` variable, the use is compliant to this rule by exception.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-0-11

The plain char type shall only be used for the storage and use of character values

## Description

### Rule Definition

*The plain char type shall only be used for the storage and use of character values.*

### Rationale

The signedness of plain `char` is implementation-defined. Because its sign is not well-defined, the plain `char` type is not suitable for use with numeric values. Use plain `char` for the storage and use of character values.

### Polyspace Implementation

The checker raises a violation when a value of signed or unsigned integer type is implicitly converted to the plain char type.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using Plain `char` For Numeric Data

```
#include<stdint.h>
typedef  char char_t;
void foo(){
char a = 'a'; // Compliant
char_t b = '\r'; // Compliant
char_t c = 10; // Noncompliant
char d = 'd'; // Compliant
}
```

In this example, Polyspace flags the use of plain `char` for numeric data.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-0-12

Signed char and unsigned char type shall only be used for the storage and use of numeric values

## Description

### Rule Definition

*Signed char and unsigned char type shall only be used for the storage and use of numeric values.*

### Rationale

In C/C++, there are three types of `char`:

- Plain `char`
- `signed char`
- `unsigned char`

The signedness of plain `char` is implementation-defined. Plain `char` cannot be interchangeably used with the other types. For instance, you might assume `char` is unsigned and use `unsigned char` to store character. Your implementation might interpret characters as signed. In such a situation, your code might behave in unexpected manner, leading to bugs that are difficult to diagnose.

MISRA C++:2008 limits the use of these three types of `char` for different applications. The `signed` and `unsigned char` type is appropriate for numeric values and storage. The plain `char` is appropriate for character data. Avoid using `signed` or `unsigned char` when you intend to use the plain `char`.

This rule also applies to the different `typedef` of these `char` types, such as `uint8_t` and `int8_t`. See `MISRA C++:2008 Rule 3-9-2`.

### Polyspace Implementation

Polyspace raises a violation of this rule when a plain `char` is implicitly converted to either `signed char` or `unsigned char`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use Plain char to Store Characters

```
typedef signed   char         int8_t;
typedef unsigned char         uint8_t;

namespace foo
{
    int8_t        ch_1   =   'a';     // Noncompliant
    uint8_t       ch_2  =   '\r';     // Noncompliant
    char          ch_3  =   'A';      // Compliant
    int8_t        num_1   =   10;     // Compliant
    uint8_t       num_2   =   12U;    // Compliant
    signed char   num_3   =   11;     // Compliant

};
```

In this example, Polyspace flags the use of `signed char` and `unsigned char` to store character data. The character literals are of plain `char` types, and Polyspace flags the implicit conversion of these plain `char` types to explicitly `signed` or `unsigned char` types.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)` | `MISRA C++:2008 Rule 3-9-2`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-0-14

The first operand of a conditional-operator shall have type bool

## Description

### Rule Definition

*The first operand of a conditional-operator shall have type bool.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-0-15

Array indexing shall be the only form of pointer arithmetic

## Description

### Rule Definition

*Array indexing shall be the only form of pointer arithmetic.*

### Rationale

You can traverse an array in two ways:

- Increment or decrement an array index, and then use the array index to access an element.
- Increment or decrement a pointer to the array, and then dereference the pointer.

The first method is clearer and less error-prone. All other forms of explicit pointer arithmetic introduce the risk of accessing unintended memory locations.

As an exception, incrementing or decrementing pointer based iterators is compliant with this rule.

### Polyspace Implementation

The checker flags:

- Arithmetic operations on all pointers, for instance `p+I`, `I+p` and `p-I`, where `p` is a pointer and `I` an integer..
- Array indexing on nonarray pointers.

Polyspace does not flag incrementing or decrementing pointer based iterators, including these standard iterator types:

- `iterator`
- `cont_iterator`
- `reverse_iterator`
- `const_reverse_iterator`

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Explicitly Calculated Pointer Value When Indexing

```
#include<vector>
template < typename IterType >
int sumValues(IterType iter, IterType end)
{
    int result = 0;
    while (iter != end) {
        result += *iter;
        ++iter;  //Noncompliant
    }
    return result;
}
int sumVec(std::vector<int>& v)
{
    int res = 0;
    for (auto it = v.begin(); it != v.end(); ++it) //Compliant by exception
    res += *it;
    return res;
}
int sumVecModern(std::vector<int>& v)
{
    int res = 0;
    for(auto i:v){
        res+=i;
    }
    return res;
}
void foo(int* p_int, int arr_int[])
{
    p_int = p_int + 1; //Noncompliant
    arr_int[0] = arr_int[1];

    p_int[5] = 0; //Noncompliant
    *(p_int + 5) = 0; //Noncompliant
    arr_int[5] = 0.0;

    int a[100];
    std::vector<int> v(100);
    sumValues(&a[0],&a[99]);

}
```

In this example, indexing is done by using array indexing and by calculating the pointer values explicitly. In the function `foo()`:

- Polyspace flags the instances where a pointer value is explicitly calculated, such as `p_int+1` or `*(p_int+5)`.
- Polyspace flags the use of array indexing on the nonarray pointer `p_int`.

Polyspace does not flag uses of array indexing on an array that is compliant with this rule.

Incrementing and decrementing iterators to containers is compliant with this rule by exception. Because `sumVec()` increments an `iterator` object, Polyspace does not flag the increment operation. This exception does not apply to raw pointers. For instance, `sumValues` is instantiated in `foo()` with `int*`. Polyspace flags incrementing the raw pointer. In modern C++, the best practice is to use range-based for loops, as shown in the function `sumVecModern()`.

## Check Information

**Group:** Expressions

**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-0-16

A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array

## Description

### Rule Definition

*A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.*

### Rationale

It is undefined behavior when the result of a pointer arithmetic operation that uses a pointer to an array element does not point to either:

- An element of the array.
- One past the last element of the array. For instance:

  ```
  int arr[3];
  int* res;
  res = arr+3; // res points to one beyond arr
  ```

The rule applies to these operations. `ptr` is a pointer to an array element and `int_exp` is an integer expression.

- `ptr` + `int_exp`
- `int_exp` + `ptr`
- `ptr` - `int_exp`
- `ptr` + +
- ++`ptr`
- --`ptr`
- `ptr`--
- `ptr` [ `int_exp` ]

### Polyspace Implementation

- Single objects that are not part of an array are considered arrays of one element. For instance, in this code example, `arr_one` is equivalent to an array of one element. Polyspace does not flag the increment of pointer `ptr_to_one` because it points to one past the last element of `arr_one`.

  ```
  void f_incr(int* x){
      int* ptr_to_one = x;
      ++ptr_to_one;  // Compliant
  }

  void func(){
      int arr_one=1; // Equivalent to array of one element
      f_incr(&arr_one);
  }
  ```

- Polyspace does not flag the use of pointer parameters in pointer arithmetic operations when those pointers point to arrays. For instance, in this code snippet, the use of `&a1[2]` in `f1` is compliant when you pass an array to `f1`.

  ```
  void f1( int* const a1){
         int* b= &a1[2]; // Compliant
  }
  void f2(){
      int arr[3] {};
      f1(arr);
  }
  ```
- In structures with multiple elements, Polyspace does not flag the result of a pointer arithmetic operation on an element that results in a pointer that points to a different element if the pointer points within the allocated memory of the structure or to one past the last element of the structure.

  For instance, in this code snippet, the assignment to `ptr_to_struct` is compliant because it remains inside `myStruct`, even if it points outside `myStruct.elem1`. Using an index larger than the element dimension to access the content of that element is not compliant, even if the resulting address is within the allocated memory of the structure.

  ```
  void func(){
      struct {
          char elem1[10];
          char elem2[10];
      } myStruct;

      char* ptr_to_struct = &myStruct.elem1[11]; //Compliant
        // Address of myStruct.elem1[11] is inside myStruct
        char val_to_struct = myStruct.elem1[11]; // Non-compliant
  }
  ```
- In multidimensional arrays, Polyspace flags any use of indices that are larger than a subarray dimension to access an element of that subarray. Polyspace does not flag the assignment of the address of that same subarray element if the address is inside the allocated memory of the top-level array.

  For example, in this code snippet, the assignment to pointer `ptr_to_arr` is compliant because the pointer points to an address that is within the allocated memory of `multi_arr`. The assignment to variable `arr_val` is not compliant because the index used to access the subarray element (3) is larger than the dimension of the subarray (2).

  ```
  void func(){
      int multi_arr[5][2];

        // Assigned memory is inside top level array
      int* ptr_to_arr = &multi_arr[2][3]; //Compliant

      // Use of index 3 with subarray of size 2
      int arr_val = multi_arr[2][3]; // Non-compliant
  }
  ```
- Polyspace flags the dereference of a pointer when that pointer points to one past the last element of an array. For instance, in this code snippet, the assignment of `ptr` is compliant, but the dereference of `ptr` is not. `tab+3` is one past the last element of tab.

  ```
  void derefPtr(){
      int tab[3] {};
  ```

```
        int* ptr = tab+3; //Compliant
        int res = *(tab+3); // Non-compliant
    }
```

- Polyspace does not raise this checker when the result of a pointer arithmetic operation results in `nullptr`. For instance, consider this code:

  ```
  void g(int *p);

  void add(int* p, int n) {
      g(p + n); //Compliant
  }

  void foo() {
      add(nullptr, 0);

  }
  ```

  The pointer arithmetic in `add()` results in a `nullptr`. Polyspace does not flag this operation.

### Extend Checker

A default Bug Finder analysis might not raise a violation of this rule when the input values are unknown and only a subset of inputs can cause an issue. To check for violations caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Pointer Arithmetic by Using Pointers to Array Elements

```
void f_incr(int* x)
{
    int* ptr_to_one = x;
    ++ptr_to_one;  // Compliant
}

void f1(int* const a1)
{
    int* b = &a1[2]; // Compliant
}

int main()
{

    int arr_one = 1; // Equivalent to array of one element
    f_incr(&arr_one);

    int arr[3] {};
    f1(arr);

    struct {
```

```
        char elem1[10];
        char elem2[10];
    } myStruct;

    char* ptr_to_struct = &myStruct.elem1[11]; // Compliant
    ptr_to_struct = &myStruct.elem2[11]; //Non-compliant

    int tab[3] {1, 2, 3};
    int* ptr =  &tab[2];
    int res = tab[2];
    ++ptr; // Compliant
    res = *ptr; //Non-compliant

    return 0;
}
```

In this example:

- The increment of `ptr_to_one` inside `f_incr()` is compliant because the operation results in a pointer that points to one past the last element of array `x`. The integer that is passed to `f_incr()` is equivalent to an array of one element.
- The operation on pointer parameter `a1` inside `f1()` is compliant because the pointer points to array `arr`.
- The first assignment of `ptr_to_struct` is compliant because `elem1[11]` is still inside `myStruct`. The second assignment of `ptr_to_struct` is not compliant because the result of the operation does not point to either inside `myStruct` or to one past the last element of `myStruct`.
- The increment of `ptr` is compliant because the result of the operation points to one past the last element of `tab`. The dereference of `ptr` on the next line is not compliant.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-0-17

Subtraction between pointers shall only be applied to pointers that address elements of the same array

## Description

### Rule Definition

*Subtraction between pointers shall only be applied to pointers that address elements of the same array.*

### Rationale

When you subtract between two pointers to elements in the same array, the result is the distance between the two array elements. If the pointers are null or point to different arrays, a subtraction operation is undefined. If you use the subtraction result as a buffer index, it can cause a buffer overflow.

Before you subtract between pointers to array elements, check that they are non-null and that they point to the same array.

### Polyspace Implementation

Polyspace flags subtraction between pointers to elements of different arrays.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Subtracting Pointers to Elements of Different Arrays

```
void foo(){
    int a[10];
    int b[10];
    int distance = a-b;//Noncompliant

}
```

In this example, Polyspace flags the subtraction between `a` and `b`, which are elements of different arrays.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-0-18

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array

## Description

### Rule Definition

*>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.*

### Rationale

When you compare two pointers to array elements, the result is the positions of the pointers relative to each other. If the pointers are null or point to different arrays, a comparison operation is undefined.

Before you use >, >=, <, or <= between pointers to array elements, check that they are non-null and that they point to the same array.

### Polyspace Implementation

Polyspace flags the use of >, >=, <, or <= operators between pointers to elements of different arrays.

The checker ignores casts when showing the violation on relational operator use with pointers types.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

### Avoid Comparing Pointers to Elements of Different Arrays

```
bool foo(){
    int a[10];
    int b[10];
    return (a<b);

}
```

In this example, Polyspace flags the comparison between a and b, which are elements of different arrays.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-0-20

Non-constant operands to a binary bitwise operator shall have the same underlying type

## Description

### Rule Definition

*Non-constant operands to a binary bitwise operator shall have the same underlying type.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-0-21

Bitwise operators shall only be applied to operands of unsigned underlying type

## Description

### Rule Definition

*Bitwise operators shall only be applied to operands of unsigned underlying type.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-2-2

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of `dynamic_cast`

## Description

### Rule Definition

*A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of `dynamic_cast`.*

### Rationale

A `virtual` base implies that multiple classes might be derived from it.

```
class Base{};
class childA: virtual public Base{};
class childB: virtual public Base{};
class childFinal: public childA, public childB{};;
```

In the preceding code, the derived classes `childA` and `childB` share the same instance of the class `Base`. The necessary pointer arithmetic is unknown at compile time when you cast from `Base*` to `childA*` or `childB*`. The offset of the child classes compared to `Base` is known only at run time.

Use `dynamic_cast` when downcasting a pointer to a virtual base class into a pointer to a child class. Using any other casting operation for this purpose results in an undefined behavior.

### Polyspace Implementation

Polyspace raises a violation if these conditions are met:

- A virtual base class is downcast.
- The casting operation is not done by using `dynamic_cast`

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use `dynamic_cast` to Downcast Pointers to `virtual` Base

```
class Base{};
class childA: virtual public Base{};
class childB: virtual public Base{};
class childFinal: public childA, public childB{};

void bar(){
    Base* pB;
    //...
    static_cast<childA*>(pB);//Noncompliant
}
```

In this example, the pointer to a `virtual` base `pB` is downcast to a pointer to a derived class. The information needed to cast `pB` to a `childA*` type is known at run time only. Using `static_cast` to perform these casts results in an undefined behavior. Polyspace raises a violation. This issue might also be reported as a compilation error.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-2-3

Casts from a base class to a derived class should not be performed on polymorphic types

## Description

### Rule Definition

*Casts from a base class to a derived class should not be performed on polymorphic types.*

### Rationale

Explicitly casting a polymorphic base class into a derived class bypasses the layers of abstraction in the implementation hierarchy, resulting in increased coupling and dependency between the classes.

When using a polymorphic class, avoid performing explicit downcasts. A pointer to the polymorphic base class does not require explicit casting into a pointer to a derived class.

### Polyspace Implementation

Polyspace raises a violation if a base class is cast to a derived class.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Downcasting In a Polymorphic Class Hierarchy

```
class Base{ //Abstract base class
    //...
public:
    virtual bool getStatus() const = 0;
};
class Derived: public Base{
    //...
public:
    virtual bool getStatus() const{ //Implementation of getStatus()
        //...
    }
};
void printStatus(const Base& B){
    const Derived* D = dynamic_cast<const Derived*>(&B);//Noncompliant
    D->getStatus();
}
void printStatus_compliant(const Base& B){
    B.getStatus();//Compliant
}
```

In this example, the `Base` class pointer `B` is cast to a `Derived` class pointer in `printStatus`. This cast is noncompliant and Polyspace raises a violation. The function `getStatus_compliant()` shows a compliant implementation of this functionality, where `virtual` functions invoke the `Derived` class member function through a pointer to the `Base` class.

## Check Information

**Group:** Expressions
**Category:** Advisory, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-2-6

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type

## Description

### Rule Definition

*A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.*

### Rationale

Because calling a function through a pointer with incompatible type results in undefined behavior, the rule forbids these cast operations:

- Casting from a function pointer to any other type.
- Casting from a function pointer to another function pointer, if the function pointers have different arguments and return types.

### Polyspace Implementation

Polyspace raises a violation of this rule if the source type of a cast operation is a pointer to a function.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Casting Function Pointers

```
void* (*func1) (int);
void* (*func2)(float);
void bar(void* (*)(int));
void foo(){

    bar(reinterpret_cast<void* (*) (int)> (func2)); //Noncompliant
    bar(func1);
}
```

In this example, the function `bar` accepts a pointer to a function that takes an `int` as an input and returns a `void*` pointer. In `foo()`, the function pointer `func2` is converted so that it can be passed to `bar`. Polyspace flags the noncompliant conversion of function pointers.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-2-8

An object with integer type or pointer to void type shall not be converted to an object with pointer type

## Description

### Rule Definition

*An object with integer type or pointer to void type shall not be converted to an object with pointer type.*

### Rationale

Converting between integral types or a `(void)*` type to other pointer types can result in unspecified behavior.

### Polyspace Implementation

The checker allows an exception on zero constants, such as `0x0`, `0`, or `0U`.

Objects with pointer type include objects with pointer-to-function type.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Converting to Pointers

In this example, Polyspace flags the conversion of integers to pointers. Polyspace does not flag converting zero constants such as `0`, `0x0`, or `0U` to pointers. Such conversions are equivalent to zero initialization of a pointer, which has a specified behavior.

```
void foo ()
{
  void *p1;
  void *p2;
  void *p3;

  p1 = static_cast < void *>(0x0);    //Compliant
  p2 = static_cast < void *>(0U);    //Compliant
  p3 = static_cast < void *>(0);    //Compliant

  p1 = (void *) 0xDEADBEEF;    //Noncompliant
  p2 = (void *) 03;       //Noncompliant
  p3 = (void *) 12345678U;    //Noncompliant
}
```

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-2-9

A cast shall not convert a pointer type to an integral type

## Description

### Rule Definition

*A cast shall not convert a pointer type to an integral type.*

### Rationale

The C++ standard specifies only the minimum size required for integral types. The implemented size of integral types and pointers depends on your hardware and development environment. In an environment where pointers have a larger size than integral types, casting a pointer type to an integral type results in an overflow. Avoid casting pointers to integral types.

### Polyspace Implementation

Polyspace flags a casting operation if it converts a pointer type variable to an integral type variable.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Casting Pointers to Integers

```
void foo(){
    int* pInt;
    //...
    int address = reinterpret_cast<int>(pInt);//Noncompliant
}
```

In this example, the pointer `pInt` is cast to an int. Depending on your environment, this operation might result in an overflow. For instance, in a 64-bit system, the size of `pInt` might be 64-bit and the size of `address` might be 32-bit. Polyspace flags this casting operation.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-2-10

The increment (++) and decrement (--) operators shall not be mixed with other operators in an expression

## Description

### Rule Definition

*The increment (++) and decrement (--) operators shall not be mixed with other operators in an expression.*

### Rationale

Using the increment and decrement operators with other operators in an expressions results in code that is difficult to read. Such code might lead to undefined behavior.

### Polyspace Implementation

Polyspace flags an expression if it contains the increment or decrement operators mixed with other operators. If an expression contains multiple increment or decrement operators mixed with other operators, Polyspace flags the first increment or decrement operator.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Mixing Increment or Decrement Operators with Other Operators

```
void foo(int a, int b){
    int c = ++a + b--; //Noncompliant
    if(--c + --a - ++b){ //Noncompliant
        //...
    }
}
```

Polyspace flags the mixing of the ++ and `--` operators with other mathematical operators in an expression.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-2-11

The comma operator, && operator and the || operator shall not be overloaded

## Description

### Rule Definition

*The comma operator, && operator and the || operator shall not be overloaded.*

### Rationale

When you overload an operator, the overloaded operator behaves as a function call. The comma operator, the && operator, and the `||` operator have certain behaviors that cannot be replicated by their overloaded counterpart. For instance, a compiler might short circuit the built-in && or `||` operators. But such short circuiting is not possible when you use an overloaded version of these operators.

Overloading these operators creates confusion about how these operators behave. Avoid overloading the comma operator, the && operator, and the `||` operator.

### Polyspace Implementation

Polyspace flags the overloading of these operators:

- Comma operator
- && operator
- `||` operator

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Do Not Overload the && Operator

```
class flag{/**/};
class Util
{
public:
flag getValue ( );
flag setValue ( int const & );
};

bool operator && ( flag const &, flag const & ); // Noncompliant
void f2 ( Util & in3, Util & in4 )
{
in3.getValue ( ) && in4.setValue ( 0 ); // Both operands evaluated
}
```

In this example, the && operator is overloaded for the class `flag`. In `f2()`, the overloaded operator is used. The overloading prevent the short circuiting. The behavior of the overloaded operator might be unexpected. Polyspace flags the overloading of the && operator.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-2-12

An identifier with array type passed as a function argument shall not decay to a pointer

## Description

### Rule Definition

*An identifier with array type passed as a function argument shall not decay to a pointer.*

### Rationale

When you pass an array to a function as a pointer, the size information of the array becomes lost. Losing information about the array size might lead to confusion and unexpected behavior.

Avoid passing arrays as pointers to function. To pass arrays into a function, encapsulate the array into a class object and pass the object to functions. Starting in C++11, the standard template library implements several container classes that can be used to pass an array to a function. C++20 has the class `std::span`, which preserves the size information.

### Polyspace Implementation

Polyspace raises a violation when you use an array in a function interface.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using Arrays in Function Interfaces

```
void f1( int p[ 10 ] ); // Noncompliant

void f2( int ( &p )[ 10 ] );// Compliant
void foo ()
{
int a[ 10 ];
f1( a );

f2( a );
}
```

In this example,the interface of `f1()` uses an array. When you pass the array `a[10]` to `f1()` as a pointer, the size of the array `a` is lost. Polyspace flags the declaration of `f1()`. If you pass arrays to function while preserving the dimensionality information, as shown by `f2()`, Polyspace does not raise a violation.

## Check Information

**Group:** Expressions

**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-3-1

Each operand of the ! operator, the logical && or the logical || operators shall have type bool

## Description

### Rule Definition

*Each operand of the ! operator, the logical && or the logical || operators shall have type bool.*

### Rationale

Use of non-`bool` operands with the logical operators `!`, && or `||`, is likely to indicate a programming error.

### Polyspace Implementation

The checker reports a violation if the operands of the logical operators `!`, && or `||`, are not effectively boolean. Allowed operands are:

- Variables of type `bool` or specified as effectively boolean using the option `Effective boolean types (-boolean-types)`.
- Expressions that compare two variables using operators such as == or < and return a boolean.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Logical && With Boolean and Non-Boolean Operands

In this example, both operands of the logical && in the `if` condition are effectively boolean and comply with the rule. The right operand of the logical && in the `else if` condition has a non-boolean type (`int32_t`) and violates the rule.

```
#include <cstdint>

class Pair
{
    int32_t leftVal;
    int32_t rightVal;
public:
    bool operator==(const Pair& anotherPair) {
        return(((*this).leftVal == anotherPair.leftVal)
             &&((*this).rightVal == anotherPair.rightVal));
    }
};

Pair getAPair();
int32_t checkPair(const Pair*);
```

```
void main() {
    Pair aPair = getAPair();
    Pair anotherPair = getAPair();
    Pair athirdPair = getAPair();

    if((aPair == anotherPair) &&
       (aPair == athirdPair)) //Compliant
       {}
    else if((aPair == anotherPair) && //Noncompliant
            (checkPair(&athirdPair)))
       {}
}
```

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-3-2

The unary minus operator shall not be applied to an expression whose underlying type is unsigned

## Description

### Rule Definition

*The unary minus operator shall not be applied to an expression whose underlying type is unsigned.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-3-3

The unary & operator shall not be overloaded

## Description

### Rule Definition

*The unary & operator shall not be overloaded.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-3-4

Evaluation of the operand to the sizeof operator shall not contain side effects

## Description

### Rule Definition

*Evaluation of the operand to the sizeof operator shall not contain side effects.*

### Polyspace Implementation

The checker does not show a warning on volatile accesses and function calls

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-8-1

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand

## Description

### Rule Definition

*The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required, Partially automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M5-14-1

The right hand operand of a logical &&, || operators shall not contain side effects

## Description

### Rule Definition

*The right hand operand of a logical &&, || operators shall not contain side effects.*

### Rationale

When evaluated, an expression that has side effects modifies at least one of the variables in the expression. For instance, `n++` is an expression with side effect.

The right operand of a:

- Logical && operator is evaluated only if the left-hand operand evaluates to true.
- Logical `||` operator is evaluated only if the left-hand operand evaluates to false.

In other cases, the right operands are not evaluated. Side effects of the expression do not take place. If your program relies on such side effects, you might see unexpected results.

### Polyspace Implementation

The checker flags logical && or `||` operators whose right operands are expressions that have side effects. Polyspace assumes:

- Expressions that modifies at least one of its variables have side effects.
- Explicit constructors or conversion functions that are declared but not defined have no side effects. Defined conversion functions have side effects.
- Volatile accesses and function calls have no side effects.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Side Effects in Right Operand of Logical Operation

```
class real32_T {
public:
    real32_T() = default;

    /* Casting operations */
    explicit real32_T(float a) {
        // ...
    }
    /* Relational operators */
    bool operator==(real32_T a) const;
    bool operator>(real32_T a) const;
};

void bar() {
    real32_T d;



    if ((d == static_cast<real32_T>(0.0F))
    || (static_cast<real32_T>(0.0F) > d)) {//Noncompliant
        /**/
    }
}



void foo(int i, int j){
    if(i==0 && ++j==i){ //Noncompliant
        --i;
    }
}
```

In the function `foo`, the right operand of the && operator contains an increment operation, which has a side effect. Polyspace flags the operator. In the function `bar`, the right operand of the `||` operator contains a conversion function that is implemented in the class. Polyspace considers such constructor to have side effects. Because the right operator has side effects, the operator is flagged.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-18-1

The comma operator shall not be used

## Description

### Rule Definition

*The comma operator shall not be used.*

### Rationale

The comma operator takes two operands. It evaluates the operators from left to right, discards the value of the left operand, and returns the value of the right operand. This operator has the lowest operator precedence. These properties can make the use of the comma operator nonintuitive. For instance, consider this code:

```
array[i++,j] = 1
```

At a glance, it might appear that the preceding code accesses a multidimensional array. In fact, this code is equivalent to:

```
i++;
array[j] = 1;
```

The use of the comma operator makes the code difficult to read and maintain. To avoid confusion, do not use the comma operator. Refactor the expression into multiple statements instead.

### Polyspace Implementation

Polyspace flags the use of comma operator. Violations are not raised when you use comma operator for function calls and initializations.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Comma Usage in C++ Code

```
typedef signed int abc, xyz, jkl;
static void func1 ( abc, xyz, jkl );       /* Compliant - case 1 */
int foo(void)
{
    volatile int rd = 1;                   /* Compliant - case 2*/
    int var=0, foo=0, k=0, n=2, p, t[10];  /* Compliant - case 3*/
    int abc = 0, xyz = abc + 1;            /* Compliant - case 4*/
    int jkl = ( abc + xyz, abc + xyz );    /* Noncompliant - case 1*/
    var = 1, foo += var, n = 3;          /* Noncompliant - case 2*/
    var = (n = 1, foo = 2);              /* Noncompliant - case 3*/
    for ( int *ptr = &t[ 0 ],var = 0 ;
          var < n; ++var, ++ptr){}    /* Noncompliant - case 4*/
    if ((abc,xyz)<0) { return 1; }        /* Noncompliant - case 5*/
}
```

In this example, the code shows various uses of commas in C code.

**Noncompliant Cases**

| Case | Reason for noncompliance |
|---|---|
| 1 | When reading the code, it is not immediately obvious what `jkl` is initialized to. |
| 2 | When reading the code, it is not immediately obvious whether `foo` has a value 0 or 1 after the statement. |
| 3 | When reading the code, it is not immediately obvious what value is assigned to `var`. |
| 4 | When reading the code, it is not immediately obvious which values control the `for` loop. |
| 5 | When reading the code, it is not immediately obvious whether the `if` statement depends on `abc`, `xyz`, or both. |

**Compliant Cases**

| Case | Reason for compliance |
|---|---|
| 1 | Using commas to call functions with variables is allowed. |
| 2 | Comma operator is not used. |
| 3 & 4 | When using the comma for initialization, the variables and their values are immediately obvious. |

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M5-19-1

Evaluation of constant unsigned integer expressions shall not lead to wrap-around

## Description

### Rule Definition

*Evaluation of constant unsigned integer expressions shall not lead to wrap-around.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Expressions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M6-2-1

Assignment operators shall not be used in sub-expressions

## Description

### Rule Definition

*Assignment operators shall not be used in sub-expressions.*

### Rationale

When used in a subexpression, assignment operators have side effects that are difficult to predict. These side effects might produce results contrary to developer expectations. This rule helps in avoiding confusion between the assignment operator (=) and the equal to operator (==). Do not use assignment operators in subexpressions.

### Polyspace Implementation

Polyspace raises this defect whenever a subexpression contains an assignment operator.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Assignment Operators in Sub-Expressions Are Noncompliant

```
#include <cstdint>

bool example(int x, int y)
{
    if (x == 10)         //Compliant
    {
        return true;
    }

       if ((x = y) == 0)            //Noncompliant
    {
        return false;
    }

    return false;
}
```

Because the assignment operator = is used in the subexpression `(x = y)`, Polyspace flags it as noncompliant.

## Check Information

**Group:** Statements

**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M6-2-2

Floating-point expressions shall not be directly or indirectly tested for equality or inequality

## Description

### Rule Definition

*Floating-point expressions shall not be directly or indirectly tested for equality or inequality.*

### Rationale

Due to the inherent rounding errors of floating-point numbers, there is no way to reliably compare floating-point numbers for equality. This includes both direct and indirect tests of equivalency. Comparisons of floating-point numbers can result in a false outcome when you expect equivalence. This behavior is unpredictable and can vary between implementations.

Avoid using floating-point numbers for equivalence comparisons. Alternatively, write a library that implements your comparison operations. Take into account the magnitude of numbers to compare as well as floating-point granularity during creation of this library.

### Polyspace Implementation

The rule checker detects the use of == or != with floating-point variables or expressions. Additionally, the rule checker detects indirect tests of equality or inequality of floating-point variables. For example:

```
float x, y = 0.0;
if ((x < y) || (x > y))           //Noncompliant
{
    //...
}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Floating-Point Number Comparisons of Equivalence

This code contains different examples of floating-point number comparisons. Polyspace reports a violation for each of these floating-point tests of equality or inequality.

- `x == y`, due to the use of ==
- `x == 0.0f`, due to the use of `!=`
- `((x <= y) && (x >= y))`, due to an indirect comparison of equality
- `((x < y) || (x > y))`, due to an indirect comparison of inequality

```
#include <cmath>
#include <limits>
```

```
void main()
{
    float x, y = 0.0;
    if (x == y)          //Noncompliant
    {
        //...
    }
    if (x != 0.0f)       //Noncompliant
    {
        //...
    }
    if ((x <= y) && (x >= y))         //Noncompliant
    {
        //...
    }
    if ((x < y) || (x > y))           //Noncompliant
    {
        //...
    }
}
```

## Check Information

**Group:** Statements
**Category:** Required, Partially automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-2-3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character

## Description

### Rule Definition

*Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.*

### Rationale

A null statement:

```
;
```

used on a line with other code can indicate a programming error. If you want to enter a null statement, place the statement on a line by itself.

A comment can follow a null statement, but use a white space character between the null statement and the comment to provide a visual cue for reviewers.

### Polyspace Implementation

The rule checker considers a null statement as a line where the first character excluding comments is a semicolon. The rule checker reports violations for situations where:

- Comments appear before the null statement.

  For instance:

  ```
  /* wait for pin */ ;
  ```
- Comments appear immediately after the semicolon without a white space in between.

  For instance:

  ```
  ;// wait for pin
  ```

The rule checker also reports a violation when a second statement appears on the same line following the null statement.

For instance:

```
; count++;
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Proper Formatting for Null Statement

In this example, there are three different versions of comments after a null statement.

- The first option is compliant. The comment comes after the null statement, the null statement is the only statement on the line, and there is a white space separating the semicolon and the comment.
- The second option is noncompliant. The comment comes before the null statement.
- The third option is noncompliant. A white space does not separate the semicolon and comment.

```
#include <iostream>

void main()
{
    std::cout << "Hello World" << std::endl;
     ; // comment here - Compliant
     /* comment here */ ; // Noncompliant
     ;//comment here - Noncompliant

}
```

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-3-1

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement

## Description

### Rule Definition

*The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.*

### Rationale

A compound statement is included in braces.

If a block of code associated with an iteration or selection statement is not contained in braces, you can make mistakes about the association. For example:

- You can wrongly associate a line of code with an iteration or selection statement because of its indentation.
- You can accidentally place a semicolon following the iteration or selection statement. Because of the semicolon, the line following the statement is no longer associated with the statement even though you intended otherwise.

This checker enforces the practice of adding braces following a selection or iteration statement even for a single line in the body. Later, when more lines are added, the developer adding them does not need to note the absence of braces and include them.

### Polyspace Implementation

The checker flags `for` loops where the first token following a `for` statement is not a left brace, for instance:

```
for (i=init_val; i > 0; i--)
   if (arr[i] < 0)
      arr[i] = 0;
```

Similar checks are performed for `switch`, `for` and `do..while` statements.

The second line of the message on the **Result Details** pane indicates which statement is violating the rule. For instance, in the preceding example, the second line of the message states that the `for` loop is violating the rule.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Statements

**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M6-4-1

An if ( condition ) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement

## Description

### Rule Definition

*An if ( condition ) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.*

### Rationale

If you use single statements as bodies of if or if...else constructs, then attempting to change the body into a compound statement might result in logic errors and unexpected results. This rule ensures that braces are not missed or forgotten when attempting to turn a single statement body into a compound statement body. Failure to use compound statements might provide unexpected results and cause developer confusion. Use `{ }` braces to create compound statements. Use braces even when the if statement is simple and contains only a single statement.

### Polyspace Implementation

Polyspace raises this defect whenever a compound statement does not immediately follow an if statement, else-if statement, or else statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Using Compound Statements in If...Else Conditionals

```
#include <cstdint>

int example(int test, int result)
{
    if (test > 5)
    {
        test--;
        result = test + result;   //Compliant
    }
    else if (test <= 5)
    {
        result = test - result;   //Compliant
    }
    else                          //Noncompliant
        result = test;

    return result;
}
```

Because the else statement does not use `{ }` braces to form a compound statement, Polyspace flags it as noncompliant.

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-4-2

All if ... else if constructs shall be terminated with an else clause

## Description

### Rule Definition

*All if ... else if constructs shall be terminated with an else clause.*

### Rationale

Ending an if... else if construct with an `else` statement is defensive programming. This final `else` statement acts as a fail-safe in case a unique situation occurs where the code progresses past the `if` and `else if` statements.

When an `if` statement is followed by one or more `else if` statements, follow the final `else if` statement with an `else` statement. Within the `else` statement provide an action. If no action is needed, provide a comment as to why no action is taken.

### Polyspace Implementation

Polyspace raises this defect whenever an `if … else if` construct does not end with an `else` statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### If Construct With No Else If Statements

```
#include <cstdint>

int example(int test, int result)
{
    if (test > 5)
    {
        test--;
        result = test + result;
    }

    return result;
}
```

Because no `else if` statement exists in this `if` construct, no final `else` statement is required within the construct.

### If Construct Containing Else-If Statements

```
#include <cstdint>
```

```
int example(int test, int result)
{
    if (test > 5)
    {
        test--;
        result = test + result;
    }
    else if (test <= 5)             //Noncompliant
    {
        result = test - result;
    }

    return result;
}
```

Because the final `else if` statement is not followed by a closing else statement, Polyspace marks it as noncompliant. Even though there should be no situation where a value of test progresses past both the `if` and `else if` statements, the additional else statement is required.

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-4-3

A switch statement shall be a well-formed switch statement

## Description

### Rule Definition

*A switch statement shall be a well-formed switch statement.*

### Rationale

In addition to the C++ standard syntax rules, MISRA defines their own syntax rules for creating well-formed switch statements. These additional syntax rules create a consistent structure for switch statements.

The additional MISRA syntax rules include:

| Rule | Syntax |
|---|---|
| switch-statement | `switch` (condition) {case-label-clause-list default-label-clause$_{opt}$} |
| case-label-clause-list | case-label case-clause$_{opt}$<br>case-label-clause-list case-label case-clause$_{opt}$ |
| case-label | `case` const-expression |
| case-clause | case-block-seq$_{opt}$ `break` ;<br>case-block-seq$_{opt}$ `throw` assignment-expression$_{opt}$ ;<br>{ statement-seq$_{opt}$ `break` ; }<br>{ statement-seq$_{opt}$ `throw` assignment-expression$_{opt}$ ; } |
| default-label-clause | default-label default-clause |
| default-label default-clause | `default` : case-clause |
| case-block | expression_statement<br>compound_statement<br>selection_statement<br>iteration_statement<br>try_block |
| case-block-seq | case-block<br>case-block-seq case-block |

These terms from the table are defined as such:

- `switch-label` — Either a `case-label` or `default-label`.
- `case-clause` — The code between any two `switch-labels`.
- `default-clause` — The code between the `default-label` and the end of the `switch` statement.
- `switch-clause` — Either a `case-clause` or a `default-clause`.

The MISRA C++ `switch` syntax rules do not include the following statements, but do permit them within the compound statements that form the body of a `switch-clause` statement.

- `labelled_statement`
- `jump_statement`
- `declaration_statement`

### Polyspace Implementation

The rule checker reports a violation in these situations:

- A statement occurs between the `switch` statement and the first `case` statement.

  For instance:

  ```
  switch(ch) {
    int temp;
    case 1:
      break;
    default:
      break;
  }
  ```
- A label or a jump statement such as `goto` or `return` occurs in the `switch` block.
- A variable is declared in a `case` statement that is outside any block.

  For instance:

  ```
  switch(ch) {
    case 1:
      int temp;
      break;
    default:
      break;
  }
  ```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Adhere to MISRA Rules for Well-Formed Switch Statements

The example fails to follow two of the well-formed `switch` statement rules from MISRA.

- The statement `int temp;` occurs between the `switch` statement and the first `case` statement.

- `case 2` contains a return statement.

Polyspace reports a single violation for the entire `switch` statement despite multiple violations within the `switch` statement.

```
int example(int x)
{
    switch (x) {            //Noncompliant
        int temp;
        case 0:
            break;
        case 1:
        case 2:
            return x;
        default:
            break;
    }

    return x;
}
```

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-4-4

A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

## Description

### Rule Definition

*A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.*

### Rationale

Placing a case-label or default-label of a switch statement in different scopes might result in unstructured code. Unstructured code might lead to unexpected behavior resulting in developer confusion. To prevent this issue, all case-labels and the default-label must be at the same scope of the compound statement forming the body of the switch statement.

### Polyspace Implementation

Polyspace raises this defect whenever a case-label belongs to any scope other than the switch statement body.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Case-Label Nested Within Another Case-Label Is Noncompliant

```
#include <cstdint>

int y, sum;

int heresASum(int a, int b)
{
    sum = a + b;
    return sum;
}

int example(int x)
{
    switch (x) {
    case 1:                         //compliant
        if (y > 0) {
        case 2:                     //noncompliant
            y = heresASum(x, y);
            break;
        }
        break;
    case 3:
        break;

    default:
        break;
    }

    return x;
}
```

Because the case-label `case 2` in nested under the `case 1` case-label, it is considered in a different scope from the other case-label statements and the default-label statement. All of these other case-label statements are in the same scope of the body of the switch-label statement.

### Check Information

**Group:** Statements
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-4-5

An unconditional throw or break statement shall terminate every non-empty switch-clause

## Description

### Rule Definition

*An unconditional throw or break statement shall terminate every non-empty switch-clause.*

### Rationale

If a throw or break statement is not used at the end of a switch-clause, control flow falls into the next switch-clause. If unintentional, this behavior might cause unexpected results. Using a throw or break statement helps to prevent unintentional fall-through behavior. Use a throw or break statement as the last statement of each case-clause and the default-clause.

Using an empty case-label is acceptable when utilizing fall-through to group together multiple clauses that otherwise require identical statements.

### Polyspace Implementation

Polyspace raises this defect whenever a case-label contains any statements and a throw or break statement is not the final statement of the case-label.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Noncompliant Unintentional Fall Through Case-Label

```
#include <cstdint>

int x, y = 2;

int example(int x)
{
    switch (x) {         //noncompliant error shows here
    case 0:              //compliant empty fall through
    case 1:
        break;           //compliant break
    case 2:
        x = y ^ 2;       //error: unintentional fall through
    case 3:              //compliant throw
        throw;

    default:             //compliant break
        break;
    }
    return x;
}
```

Because `case 2` does not contain a throw or break statement, it falls over into `case 3`. This type of fall through is noncompliant.

Because it is an empty case-label, `Case 0` will fall through to `case 1`. This is a compliant empty case-label fall through.

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-4-6

The final clause of a `switch` statement shall be the default-clause

## Description

### Rule Definition

*The final clause of a `switch` statement shall be the default-clause.*

### Rationale

It is a good defensive programming technique to always use a `default` clause as the final clause of a `switch` statement. The `default` clause must either take appropriate action or contain a comment as to why the `default` clause takes no action.

### Polyspace Implementation

The rule checker detects `switch` statements that do not have a final `default` clause.

The rule checker does not report a violation if the `switch` variable is an enumeration with finite number of values and you have a `case` clause for each value. For instance:

```
enum Colors { RED, BLUE, YELLOW } color;

switch (color) {
      case RED:
        break;
      case BLUE:
        break;
      case YELLOW:
        break;
}
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Example

### Use Default Clause as Final `Switch` Clause

Polyspace flags both `switch` statements in this example as noncompliant.

- `switch (i)` contains multiple `case` statements. However, there is no `default` clause to handle cases where `i` is any value other than 0, 1, or 2. Even if `i` can only be 0, 1, or 2, using a `default` clause is defensive programming to catch any circumstance where `i` is unexpectedly a different value.
- `switch (colors)` uses an enumeration for the `switch` variable. However, each value does not have an associated `case` clause. Adding the `case` clause for `case BLUE` makes this `switch` statement compliant.

```
#include <cstdint>

int main(){
    int i;

    switch (i)        //Noncompliant
    {
    case 0:
        break;
    case 1:
    case 2:
        break;
    }

    enum Colors { RED, BLUE, YELLOW } colors;

    switch (colors) {    //Noncompliant
    case RED:
        break;
    case YELLOW:
        break;
    }

    return 0;
}
```

### Check Information

**Group:** Statements
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-4-7

The condition of a switch statement shall not have bool type

## Description

### Rule Definition

*The condition of a switch statement shall not have bool type.*

### Rationale

Switch statements that have a bool condition might cause confusion or mistakes not caught by the compiler. If statements are better suited to handling bool evaluations. Use If...else statements in place of switch statements that have a bool condition.

### Polyspace Implementation

Polyspace raises this defect whenever a switch-case conditional results in a bool.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Using Bools in Switch Conditions Is Noncompliant

```
#include <cstdint>

int x = 10;

int example(int x)
{
    switch (x > 0) { //noncompliant
    case true:
        x += 10;
        break;

    case false:
        x -= 10;
        break;

    default:
        x = 0;
        break;
    }

    return x;
}
```

Because the switch statement condition `x > 0` results in a bool, Polyspace marks it as noncompliant.

Use the following if statement in place of the above switch-case:

```
#include <cstdint>

int x = 10;

int example(int x)
{
    if (x > 0) {
        x += 10;
    } else if (x <= 0) {
        x -= 10;
    } else {
        x = 0;
    }

    return x;
}
```

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-5-2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=

## Description

### Rule Definition

*If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.*

### Rationale

If the loop counter increments or decrements by more than one per iteration, avoid using == or != in the loop termination condition. It is unclear from visual inspection if such loop termination conditions are ever satisfied.

For instance, if a loop counter `ctr` increments by more than one per iteration, such as `ctr+=2`, and the termination condition is `ctr == someVal`, the counter might skip the value `someVal`, resulting in a nonterminating loop.

Nonterminating loops can result in developer confusion or unexpected values.

### Polyspace Implementation

Polyspace raises this defect whenever all the following are true:

- The loop-counter is not modified by `--` or `++`
- The loop-condition does not contain <=, <, > or >=

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Nonterminating Loops Created by Using Noncompliant Operand Pairings

```
#include <cstdint>

uint32_t row, col;

void example()
{
    for (row = 0; row <= 10; row++) {                 //Compliant
        for (col = 100; col != 10; col -= 4) {        //Noncompliant
            //...
        }
    }
    for (row = 0; row != 10; row++) {            //Compliant
        //...
    }
}
```

Because the second for loop-counter is not modified by `--` or `++`, `!=` becomes a noncompliant operand. Use one of the following operands instead: <=, <, > or >=.

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M6-5-3

The loop-counter shall not be modified within condition or statement

## Description

### Rule Definition

*The loop-counter shall not be modified within condition or statement.*

### Rationale

The `for` loop has a specific syntax for modifying the loop counter. A code reviewer expects modification using that syntax. Modifying the loop counter elsewhere can make the code harder to review.

### Polyspace Implementation

The checker flags modification of a `for` loop counter in the loop body or the loop condition (the condition that is checked to see if the loop must be terminated).

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-5-4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop

## Description

### Rule Definition

*The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.*

### Rationale

If the loop-counter modifier of the loop is not constant, the loop might end unpredictably, causing results contrary to your expectations. Constant increments create predictable loop termination.

### Polyspace Implementation

Polyspace reports this defect whenever the modifier of a loop-counter is not constant. Polyspace also reports this defect when you modify a loop by -=*n* or +=*n* and modify *n* within the loop.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant Nonconstant Loop-Counter Modifiers

```
#include <cstdint>

int i, x = 0;
int y, z = 1;

int ex2()
{
    return z;
}

void example()
{
    for (i = 0; i <= 10; i += x) {          //Noncompliant
        x += 1;
    }

    for (i = 0; i <= 10; i += y) {          //Compliant
        z += i;
    }

    for (i = 0; i >= 10; i -= ex2()) {      //Noncompliant
        z += i;
    }
}
```

Because the first `for` loop uses the loop-counter modifier of `+=x` and modifies the variable `x` within the `for` loop, Polyspace flags the `for` loop as noncompliant.

The second `for` loop shows a compliant case of using a loop-counter modifier of `+=y`, where `y` is not modified within the loop.

The third `for` loop modifies the loop counter by using the return value of a function, `ex2()`. Because the function returns a different value in each iteration, the loop-counter modifier is not constant. Polyspace flags the `for` loop as noncompliant.

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-5-5

A loop-control-variable other than the loop-counter shall not be modified within condition or expression

## Description

### Rule Definition

*A loop-control-variable other than the loop-counter shall not be modified within condition or expression.*

### Rationale

Loops are easier to understand and predict if loop-control-variables, other than loop counters, are not modified within the condition or increment/decrement expression. A volatile typed loop-control-variable is an exception and you can modify it outside of the statement without triggering this violation.

Loop-control-variables are any variables that occur in the loop init-statement, condition, or expression. Loop-control-variables include loop-counters and flags used for early loop termination. A loop-counter is a loop-control-variable that is:

- Initialized prior to the `for` loop or in the init-statement of the `for` loop.
- An operand to a relational operator in the condition of the `for` loop.
- Modified in the expression of the `for` loop.

### Polyspace Implementation

Polyspace raises this defect whenever a loop-counter or flag used for early termination is modified within the condition or expression.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Modifying Loop-Control-Variables Within the Loop Condition

```
#include <cstdint>

bool tf_test, tf_test2 = false;
int32_t x = 0;

bool a_test(bool a)
{
    if (x % 2 == 0)
    {
        a = true;
    }
    else
    {
        a = false;
    }
    return a;
}

void example()
{

    for (x = 0; (x < 10) && (tf_test = a_test(tf_test)); x++)        //noncompliant
    {
        //...
    }

    for (x = 0; (x < 10) && tf_test2; x++)         //compliant
    {
             //...
        tf_test2 = a_test(tf_test2);
    }
}
```

In the first `for` loop, because `tf_test` is a flag used for early loop termination that is modified within the condition, Polyspace flags it as noncompliant.

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-5-6

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool

## Description

### Rule Definition

*A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.*

### Rationale

Loops terminate when the loop-counter value meets a termination condition. You can use additional loop-control-variables if you want to end a loop early.

For instance:

`for(ctr = 0 ; ctr <= 10; ctr++) {…}` terminates when the value of `ctr` is greater than `10`.

`for(ctr = 0 ; ctr <= 10 && level > 0; ctr++) {…}` terminates when the value of `ctr` is greater than `10` or when the value of `level` is greater than `0`.

In the second instance, it is not clear why the condition `level >= 0` terminates the loop early. By using a Boolean variable as a loop-control-variable for early termination, you can use a more descriptive name that reflects the early termination state.

For example:

```
for(ctr = 0 ; ctr <= 10 && fuelTankNotEmpty; ctr++)
{
    /...
    fuelTankNotEmpty = (level >= 0);
}
```

This Boolean variable is called a flag. Boolean flags make loop control logic easier to understand.

### Polyspace Implementation

Polyspace raises this defect whenever a non-Boolean loop-control-variable is modified within the loop statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Modifying Non-Boolean Loop-Control-Variables Within the Loop Statement

```
#include <cstdint>
```

```
int32_t ctr, level = 1;
bool fuelTankNotEmpty = true;

void example()
{
    for(ctr = 0 ; ctr <= 10 && level >= 0; ctr++)        //Noncompliant
    {
        level--;
    }

    for (ctr = 0; ctr <= 10 && fuelTankNotEmpty; ctr++)    //Compliant
    {
        level--;
        fuelTankNotEmpty = (level >= 0);
    }

}
```

In the first `for` loop, because `level` is not a Boolean and is modified within the statement, Polyspace flags it as noncompliant.

The second loop shows how to use a Boolean flag to be compliant with this rule.

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-6-1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement

## Description

### Rule Definition

*Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.*

### Rationale

Using a goto statement to jump into nested blocks creates complex control flow, which might cause developer confusion or unexpected results. To avoid unexpected results, place the label the goto statement is referring to in the same block or in a block that encloses the goto statement.

### Polyspace Implementation

Polyspace raises this defect when the goto destination is in a different block than the goto statement. This defect is not raised if the goto destination is in a block enclosing the goto statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using goto Statements to Exit Out of a Block

```
#include <iostream>

int x, y = 0;

void foo1()
{
    int i = 0;
    if (x <= 10) {
        goto err;                    //Noncompliant
    }

    if(x > 10) {
    err:
        std::cout << "Error encountered in loop" << std::endl;
    }

}

void foo2()
{
    for (x = 0; x < 100; ++x) {
        for (y = 0; y < 100; ++y) {
            if (x > y) {
                goto stop;            //Compliant
            }
            //...
        }
    }
stop:
    std::cout << "Error encountered in loop" << std::endl;
}
```

Because the label `err` is located within a separate code block than `goto err`, and that code block does not enclose the code block where `goto err` resides, Polyspace flags the `goto` statement as noncompliant.

The label `stop` is not in the same block as `goto stop`, but is in a block enclosing the `goto stop` statement. This behavior is compliant behavior.

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M6-6-2

The goto statement shall jump to a label declared later in the same function body

## Description

### Rule Definition

*The goto statement shall jump to a label declared later in the same function body.*

### Rationale

Using a `goto` statement to jump to a label earlier in the same function body creates an iteration. Avoid creating iterations by using `goto` statements. Use iteration statements defined by the core language because they are easier to understand and maintain than goto statements.

### Polyspace Implementation

Polyspace raises this defect when an iteration is formed by using `goto` statements.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Creating Iterations by Using goto Statements

```
#include<iostream>
void foo(int x, int y)
{
test:
    x++;
    if (x <= y) {
        std::cout << x << std::endl;
        goto test;   //Noncompliant
    }
}
```

Because `goto test` sends the code back to the beginning of the function, the goto statement creates a loop. Use looping statements such as `while` or `for` instead.

```
#include<iostream>
void foo(int x, int y)
{
    while (x < y)
    {
        x++;
        std::cout << x << std::endl;
    }
}
```

## Check Information

**Group:** Statements
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M6-6-3

The `continue` statement shall only be used within a well-formed `for` loop

## Description

### Rule Definition

*The `continue` statement shall only be used within a well-formed `for` loop.*

### Rationale

Using a `continue` statement can add unnecessary complexity to code, which can cause issues during testing due to the additional logic required.

### Polyspace Implementation

The rule checker reports a violation for the use of `continue` statements in:

- `for` loops that are not well-formed, that is, `for` loops that violate rules 6-5-x
- `while` loops

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Example

### Only Use `continue` Statement in Well-Formed `for` Loops

The `for` loop in this code is not well-formed as it breaks `MISRA C++:2008 Rule 6-5-2`. Because the `for` loop in the example is not well-formed and contains a `continue` statement, Polyspace raises this violation.

```
#include <cstdint>

uint32_t row, col;

void example()
{
    for (row = 0; row <= 10; row++) {
        for (col = 100; col != 10; col -= 4) {
            //...
            continue;        //Noncompliant
        }
    }
}
```

## Check Information

**Group:** Statements

**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M7-1-2

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified

## Description

### Rule Definition

*A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.*

### Polyspace Implementation

The checker flags pointers where the underlying object is not const-qualified but never modified in the function body.

If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. Pointers that point to these variables are not flagged.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M7-3-1

The global namespace shall only contain main, namespace declarations and extern "C" declarations

## Description

### Rule Definition

*The global namespace shall only contain main, namespace declarations and extern "C" declarations.*

### Rationale

The rule makes sure that all names found at global scope are part of a namespace. Adhering to this rule avoids name clashes and ensures that developers do not reuse a variable name, resulting in compilation/linking errors, or shadow a variable name, resulting in possibly unexpected issues later.

### Polyspace Implementation

Other than the `main` function, the checker flags all names used at global scope that are not part of a namespace.

The checker does not flag names at global scope if they are declared in `extern "C"` blocks (C code included within C++ code). However, if you use the option `Ignore link errors (-no-extern-c)`, these names are also flagged.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declaration<br>
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M7-3-2

The identifier main shall not be used for a function other than the global function main

## Description

### Rule Definition

*The identifier main shall not be used for a function other than the global function main.*

### Rationale

Typically, the `main` function lives in the global namespace and acts as the entry point to a program. The use of `main` in other contexts might defy developer expectations.

### Polyspace Implementation

The rule checker reports a violation if you use the identifier `main` in a namespace other than the global namespace.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Nonstandard Uses of `main`

```
#include <cstdint>

int32_t main() { //Compliant
    //...
    return 0;
}

namespace Backups {
    int32_t main() { //Noncompliant
        //...
        return 0;
    }
}
```

The use of `main` in a namespace other than the global namespace violates the rule.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M7-3-3

There shall be no unnamed namespaces in header files

## Description

### Rule Definition

*There shall be no unnamed namespaces in header files.*

### Rationale

According to the C++ standard, names in an unnamed namespace, for instance, `aVar`:

```
namespace {
   int aVar;
}
```

have internal linkage by default. If a header file contains an unnamed namespace, each translation unit with a source file that `#include`-s the header file defines its own instance of objects in the namespace. The multiple definitions are probably not what you intended and can lead to unexpected results, unwanted excess memory usage, or inadvertently violating the one-definition rule.

### Polyspace Implementation

The checker flags unnamed namespaces in header files.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Unexpected Results from Unnamed Namespaces in Header Files

Header File: `aHeader.h`

```
namespace { //Noncompliant
   int aVar;
}
```

First source file: `aSource.cpp`

```
#include "aHeader.h"
#include <iostream>

void setVar(int arg) {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = arg;
    std::cout << "Value set at: " << aVar << std::endl;
}
```

Second source file: `anotherSource.cpp`

```
#include "aHeader.h"
#include <iostream>

extern void setVar(int);

void resetVar() {
    std::cout << "Current value: " << aVar << std::endl;
    aVar = 0;
    std::cout << "Value set at: 0" << std::endl;
}

void main() {
    setVar(1);
    resetVar();
}
```

In this example, the noncompliant unnamed namespace leads to two definitions of `aVar` in the translation unit from `aSource.cpp` and the translation unit from `anotherSource.cpp`. The two definitions can lead to possible unexpected output:

```
Current value: 0
Value set at: 1
Current value: 0
Value set at: 0
```

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M7-3-4

Using-directives shall not be used

## Description

### Rule Definition

*using-directives shall not be used.*

### Rationale

`using` directives can inadvertently expose more variables to name lookup than you intend. The wider scope for name lookup increases the likelihood of a variable being unintentionally used. For instance:

```
namespace NS {
    int i;
    int j;
}
using namespace NS;
```

exposes both `NS::i` and `NS::j` to name lookup, but you might have intended to use only one of the variables.

It is preferable to only expose variables that you intend to use or refer to variables by their fully qualified name when you use them. For instance, in the previous example, if you use the variable `NS::i`, expose only this variable to name lookup:

```
using NS::i;
```

or refer to `NS::i` instead of `i` in all instances.

### Polyspace Implementation

The rule checker reports a violation on `using` directives. These directives contain the keyword `using` followed by the keyword `namespace` and the name of a namespace. The `using` directives make all members of a namespace available to the current scope.

The checker does not flag `using` declarations. These declarations also contain the keyword `using` but not the keyword `namespace`. The `using` declarations only expose specific members of a namespace.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Namespace Exposed Through `using` Directive

```
namespace currentDataModel {
    int nodes;
    int count;
}
using namespace currentDataModel; //Noncompliant

namespace lastDataModel {
    int nodes;
    int count;
}
using lastDataModel::count; //Compliant
```

In this example, the `using` directive that exposes all members of the namespace `currentDataModel` violates the rule.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M7-3-6

Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files

## Description

### Rule Definition

*using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.*

### Rationale

If `using` directives or declarations are present in a header file, the order in which the header is included might affect the final program results. It is a good practice to ensure that the program behavior does not depend on the order of inclusion of headers.

### Polyspace Implementation

The checker flags `using` directives or declarations in header files.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### `using` Directives in Headers Introduce Dependence on Order of Inclusion

Header File: `headerUnits.h`:

```
void checkValueForOverflows(int32_t);

namespace Units {
    void checkValueForOverflows(int8_t);
}

inline void convertToAcceptedUnit(int8_t originalValue) {
    checkValueForOverflows(originalValue);
}
```

Header File: `headerAcceptedUnits.h`:

```
namespace Units {

}

using namespace Units; //Noncompliant
```

Source file that includes previous two headers:

```
#include <cstdint>
#include "headerUnits.h"
```

```
#include "headerAcceptedUnits.h"

int8_t convert(int8_t meterReading) {
    convertToAcceptedUnit(meterReading);
}
```

In this example, the `using` directive in the header file `headerAcceptedUnits.h` exposes the namespace `Units`. This directive plus the inlined function in the header makes the order of inclusion of `headerUnits.h` and `headerAcceptedUnits.h` important for the final program execution.

- If `headerAcceptedUnits.h` is included before `headerUnits.h`, the function `convertToAcceptedUnit()` invokes the function `checkValueForOverflows(int32_t)`.
- If `headerAcceptedUnits.h` is included after `headerUnits.h`, as shown above, the function `convertToAcceptedUnit()` invokes the function `checkValueForOverflows(int8_t)`.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M7-4-2

Assembler instructions shall only be introduced using the asm declaration

## Description

### Rule Definition

*Assembler instructions shall only be introduced using the asm declaration.*

### Rationale

The `asm` declaration is available in all C++ implementations[23] but other ways of introducing assembler instructions might not be available. Using the `asm` declaration makes your code portable across implementations.

### Polyspace Implementation

The rule checker reports uses of methods other than `asm` declarations as rule violations.

For instance, the rule checker allows the `asm` declaration:

```
asm("ADD a0,1")
```

But if the same assembler instructions are introduced through a `#pragma asm`, the checker reports a violation:

```
#pragma asm
   "ADD a0,1"
#pragma endasm
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Introduction of Assembler Instructions Using Nonportable Methods

```
void Delay1 ( void ) {
#pragma asm             //Noncompliant
   "ADD a0,1"
#pragma endasm
     ;
}

void Delay2 ( void ) {
   asm("ADD a0,1");     //Compliant
}
```

The use of `#pragma asm` violates this rule.

[23] How the `asm` declaration is implemented might vary across compilers. See also `AUTOSAR C++14 Rule A7-4-1`.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M7-4-3

Assembly language shall be encapsulated and isolated

## Description

### Rule Definition

*Assembly language shall be encapsulated and isolated.*

### Rationale

Assembler instructions can be difficult to port across implementations. Encapsulating and isolating them in a function helps to make your code portable. You can easily track down assembler instructions or provide alternative implementations without changing the rest of the code.

### Polyspace Implementation

The checker flags `asm` statements unless they are encapsulated in a function call.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Assembler Instructions Not Encapsulated in Function Call

```
void DoSomething(void);

void Delay ( void ) {
    asm( "NOP"); //Compliant
    asm( "NOP"); //Compliant
}
void DoSomethingAndDelay (void) {
    DoSomething();
    Delay();// Assembler is encapsulated
    DoSomething();
    asm("NOP"); //Noncompliant
    DoSomething();
}
```

The `asm` statement in the function `DoSomethingAndDelay()` is mixed with regular C code and violates the rule. The `asm` statements in the function `Delay()` are compliant because they form the only instructions in this function. The `Delay()` function thus effectively encapsulates the assembler instructions.

## Check Information

**Group:** Declaration
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M7-5-1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function

## Description

### Rule Definition

*A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.*

### Rationale

Local objects of a function are destroyed at the end of a function call. If you return the address of such an object via a pointer, you might access a destroyed object. For instance:

```
int* foo(int a){
    int b = a^2;
    return &b;
}
```

The function `foo()` returns a pointer to `b`. The variable `b` goes out of scope at the end of the function call and the pointer returned by `foo()` points to a destroyed object. Accessing the pointer leads to undefined behavior.

### Polyspace Implementation

Polyspace raises a violation if a function returns a reference or pointer to a variable that goes out of scope at the end of the function call.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Returning Pointers to Local Variables

```
int gVal;
//...

int* foo(void){
    int val;
    //...
    return &val;//Noncompliant
}
```

In this example, the function `foo()` returns the address of `val`. This local variables goes out of scope at the end of the function call. Accessing the pointer returned by `foo()` results in undefined behavior. Polyspace raises a violation.

## Check Information

**Group:** Declaration
**Category:** Required, Non-automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M7-5-2

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist

## Description

### Rule Definition

*The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.*

### Rationale

If an object continues to point to another object *after* the latter object ceases to exist, dereferencing the first object leads to undefined behavior.

### Polyspace Implementation

The checker flags situations where the address of a local variable is assigned to a pointer defined at global scope.

The checker does not raise violations of this rule if:

- A function returns the address of a local variable. `AUTOSAR C++14 Rule M7-5-1` covers this situation.
- The address of a variable defined at block scope is assigned to a pointer that is defined with greater scope, but not global scope.

  For instance:

  ```
   void foobar ( void )
    {
      char * ptr;
      {
        char var;
        ptr = &var;
      }
    }
  ```

  Only if the pointer is defined at global scope is a rule violation raised. For instance, the rule checker flags the assignment here:

  ```
  char * ptr;
  void foobar ( void )
    {
        char var;
        ptr = &var;
    }
  ```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Address of Local Variable Assigned to Global Pointer

```
char * ptr;

void foo (void) {
    char varInFoo;
    ptr = &varInFoo; //Noncompliant
}

void bar (void) {
    char varInBar = *ptr;
}

void main() {
    foo();
    bar();
}
```

The assignment `ptr = &varInFoo` is noncompliant because the global pointer `ptr` might be dereferenced outside the function `foo`, where the variable `varInFoo` is no longer in scope. For instance, in this example, `ptr` is dereferenced in the function `bar`, which is called after `foo` completes execution.

## Check Information

**Group:** Declaration
**Category:** Required, Non-automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M8-0-1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively

## Description

### Rule Definition

*An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.*

### Rationale

Init-declarator-lists that have multiple declarators might result in ambiguous type assignments and might cause a developer to assign unintended types to certain identifiers. Having a single init-declarator for each declaration clarifies the declaration type and reduces the risk of unwanted type assignments. Avoid multiple declarators in an init-declarator-list or a member-declarator-list.

### Polyspace Implementation

Polyspace flags declarators after the first declarator in an init-declarator-list or a member-declarator-list.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Multiple Declarators Within an Init-Declarator-List

```
#include <cstdint>
#include <string>

class exampleClass{};

void foo() {
    double a, b, c; //Noncompliant
    exampleClass objectOne, objectTwo; //Noncompliant
    int32_t d; int32_t e; //Compliant
    std::string f;  //Compliant
}
```

In this example, Polyspace flags init-declarator-lists that have multiple declarators. For instance:

- `double a, b, c` is noncompliant because the init-declarator-list consists of more than one init-declarator and the `b` and `c` declarators are flagged.
- `exampleClass objectOne, objectTwo` is noncompliant because the init-declarator-list consists of more than one init-declarator and the `objectTwo` declarator is flagged.

- `int32_t d, int32_t e` and `std::string f` are compliant because each init-declarator-list consists of a single init-declarator.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M8-3-1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments

## Description

### Rule Definition

*Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M8-4-2

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration

## Description

### Rule Definition

*The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.*

### Polyspace Implementation

The checker detects mismatch in parameter names between:

- A function declaration and the corresponding definition.
- Two declarations of a function, provided they occur in the same file.

  If the declarations occur in different files, the checker does not raise a violation for mismatch in parameter names. Redeclarations in different files are forbidden by `AUTOSAR C++14 Rule M3-2-3`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M8-4-4

A function identifier shall either be used to call the function or it shall be preceded by &

## Description

### Rule Definition

*A function identifier shall either be used to call the function or it shall be preceded by &.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M8-5-2

Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures

## Description

### Rule Definition

*Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.*

### Rationale

The use of nested braces in initializer lists to match the structures of nested objects in arrays, unions, and structs encourages you to consider the order of initialization of complex data types and makes your code more readable. For example, the use of nested braces in the initialization of `ex1` makes it easier to see how the nested arrays `arr1` and `arr2` in struct `ex1` are initialized.

```
struct Example
{
    int num;
    int arr1[2];
    int arr2[3];
};

//....
struct Example ex1 {1, {2, 3}, {4, 5, 6}}; //Compliant
```

The rule does not require the use of nested braces if you zero initialize an array, a union, or a struct with nested structures are the top-level, for instance:

```
struct Example ex1 {}; //Compliant
```

### Polyspace Implementation

If you non-zero initialize an array, union, or struct that contains nested structures and you do not use nested braces to reflect the nested structure, Polyspace flags the first element of the first nested structure in the initializer list. For instance, in this code snippet, Polyspace flags the number `2` because it corresponds to the first element of nested structure `arr1` inside struct `ex1`.

```
struct Example
{
    int num;
    int arr1[2];
    int arr2[3];
};

//....
struct Example ex1 {1, 2, 3, 4, 5, 6}; // Non-compliant
```

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Missing Nested Braces in Initializer of Two-Dimensional Arrays

```
char arr1[2][3] {'a', 'b', 'c', 'd', 'e', 'f'}; //Non-compliant
char arr2[2][3] {{'a', 'b', 'c'}, {'d', 'e', 'f'}}; //Compliant
char arr_top_level[2][3] { }; //Compliant
char arr_sub_level[2][3] { {}, {'d', 'e', 'f'}}; //Non-compliant
```

In this example, two-dimensional array `arr1` is non-compliant because the initializer list does not reflect the nested structure of this array (two arrays of three elements each). The initialization of `arr2` uses nested braces to reflect the nested structure of the array and is compliant. Similarly, the initialization of `arr_top_level` is compliant because it zero initializes the array at the top level. Note that the initialization of `arr_sub_level` is non-compliant because zero-initializes only the first sub-array while explicitly initializing all the elements of the other sub-array.

## Check Information

**Group:** Declarators
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M9-3-1

`Const` member functions shall not return non-`const` pointers or references to class-data

## Description

### Rule Definition

*`const` member functions shall not return non-`const` pointers or references to class-data.*

### Rationale

A `const` object cannot be changed post initialization and can only invoke class member functions that are also declared as `const`. These member functions are not expected to change the state of the object.

If a `const` member function returns a non-`const` pointer or reference to a data member of the class, the function can modify the state of an object and violate developer expectations. To avoid this situation, when you define a `const` member function that returns a reference or a pointer to a class data member, specify the return type as `const`.

### Polyspace Implementation

Polyspace flags a violation of this rule only if a `const` member function returns a non-`const` pointer or reference to a non-`static` data member. The rule does not apply to `static` data members.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Specify Return Type of `const` Member Function as `const`

```
class A{

    private:

    int& rInt;
    public:

    int* getR() const{ //Noncompliant
        return &rInt;
    }

    const int* getConstR() const{ //Compliant
        return &rInt;
    }
};
```

In this example:

- The `const` member function `getR()` returns a non-`const` pointer and violates the rule.
- The `const` member function `getConstR()` returns a `const` pointer and is compliant with the rule.

### Check Information

**Group:** Classes
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**
“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M9-3-3

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const

## Description

### Rule Definition

*If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.*

### Rationale

`const` member functions cannot modify the data members of the class. `static` member function cannot modify the nonstatic data members of the class. If a member function does not need to modify the nonstatic data members of the class, limit their access to data by declaring the member functions as `const` or `static`. Such declaration clearly expresses and enforces the design intent. That is, if you inadvertently attempt to modify a data member through a `const` member function, the compiler catches the error. Without the `const` declaration, this kind of inadvertent error might lead to bugs that are difficult to find or debug.

### Polyspace Implementation

The checker performs these checks in this order:

1. The checker first checks if a class member function accesses a data member of the class. Functions that do not access data members can be declared static.
2. The checker then checks functions that access data members to determine if the function modifies any of the data members. Functions that do not modify data members can be declared const.

A violation on a `const` member function means that the function does not access a data member of the class and can be declared static.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Explicitly Restrict Access for Member Functions that Do Not Modify Data Members

```
#include<cstdint>
void Connector(void);
class A
{
public:
    int16_t foo ( ) // Noncompliant
    {
        return m_i;
    }
    int16_t foo2 ( ) // Noncompliant
    {
        Connector();// Might have side-effect
        return m_i;
    }
    int16_t foo3 ( ) // Noncompliant
    {
        return m_s;
    }
    int16_t inc_m ( ) // Compliant
    {
        return ++m_i;
    }
    int16_t& getref()//Noncompliant
    {
        return m_i_ref;
    }
private:
    int16_t m_i;
    static int16_t m_s;
    int16_t& m_i_ref;
};
```

In this example, Polyspace flags the functions `foo`, `foo2`, `foo3`, and `getref` as noncompliant.

- The functions `foo` and `foo3` do not modify any nonstatic data members. Because their data access is not explicitly restricted by declaring them as `const`, Polyspace flags these functions. To fix these defects, declare `foo` and `foo3` as `const`.
- The function `foo2` does not explicitly modify any of the data members. Because it is not declared as `const`, Polyspace flags the function. `foo2` calls the global function `Connector`, which might have side effects. Do not declare `foo2` as a `const` function. In C++11 or later, `const` member functions are expected to be thread-safe, but `foo2` might not be thread-safe because of the side effects of `Connector`. To avoid data races, keep `foo2` as a nonconst function. Justify the defect by using review information or code comments.
- The function `getref` does not modify any data members. Because it is not declared as `const`, Polyspace flags it. Declaring `getref` as `const` resolves this defect, but that is not enough to restrict write access of `getref` because it returns a nonconst reference to `m_i_ref`. To restrict `getref` from modifying `m_i_ref`, the return type of `getref` must also be `const`.

## Check Information

**Group:** Classes
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M9-6-4

Named bit-fields with signed integer type shall have a length of more than one bit

## Description

### Rule Definition

*Named bit-fields with signed integer type shall have a length of more than one bit.*

### Rationale

Variables that have signed integer bit-field types of length one bit might have values that do not meet developer expectations. For instance, signed integer types of a fixed width such as `std16_t` have a two's complement representation. In this representation, a single-bit variable has just the sign bit and the variable value might be 0 or -1.

### Polyspace Implementation

The checker flags declarations of named variables having signed integer bit-field types of length equal to one.

Bit-field types of length zero are not flagged.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Compliant and Noncompliant Bit-Field Types

```
#include <cstdint>

typedef struct
{
   std::uint16_t IOFlag :1;    //Compliant - unsigned type
   std::int16_t InterruptFlag :1; //Noncompliant
   std::int16_t Register1Flag :2; //Compliant - Length more than one bit
   std::int16_t : 1; //Compliant - Unnamed
   std::int16_t : 0; //Compliant - Unnamed
   std::uint16_t SetupFlag :1; //Compliant - unsigned type
} InterruptConfigbits_t;
```

In this example, only the second bit-field declaration is noncompliant. A named variable is declared with a signed type of length one bit.

## Check Information

**Group:** Classes
**Category:** Required, Automated

## Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M10-1-1

Classes should not be derived from virtual bases

## Description

### Rule Definition

*Classes should not be derived from virtual bases.*

### Rationale

The use of virtual bases can lead to many confusing behaviors.

For instance, in an inheritance hierarchy involving a virtual base, the most derived class calls the constructor of the virtual base. Intermediate calls to the virtual base constructor are ignored.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of Virtual Bases

```
class Base {};
class Intermediate: public virtual Base {}; //Noncompliant
class Final: public Intermediate {};
```

In this example, the rule checker raises a violation when the `Intermediate` class is derived from the class `Base` with the `virtual` keyword.

The following behavior can be a potential source of confusion. When you create an object of type `Final`, the constructor of `Final` directly calls the constructor of `Base`. Any call to the `Base` constructor from the `Intermediate` constructor are ignored. You might see unexpected results if you do not take into account this behavior.

## Check Information

**Group:** Derived Classes
**Category:** Advisory, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M10-1-2

A base class shall only be declared virtual if it is used in a diamond hierarchy

## Description

### Rule Definition

*A base class shall only be declared virtual if it is used in a diamond hierarchy.*

### Rationale

This rule is less restrictive than `AUTOSAR C++14 Rule M10-1-1`. Rule M10-1-1 forbids the use of a virtual base anywhere in your code because a virtual base can lead to potentially confusing behavior.

Rule M10-1-2 allows the use of virtual bases in the one situation where they are useful, that is, as a common base class in diamond hierarchies.

For instance, the following diamond hierarchy violates rule M10-1-1 but not rule M10-1-2.

```
class Base {};
class Intermediate1: public virtual Base {};
class Intermediate2: public virtual Base {};
class Final: public Intermediate1, public Intermediate2 {};
```

### Polyspace Implementation

Polyspace flags a class if the class is used as a virtual base in a single and linear class hierarchy. In such a hierarchy, `virtual` bases are unnecessary.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using `virtual` Base Classes in Linear Hierarchy

```
class A{};
class B: public virtual A{};
class C: public virtual B{}; //Noncompliant

class alpha{};

class beta: public virtual alpha{};
class gamma: public virtual alpha{};
class omega: public beta, public gamma{}; //Compliant
```

In this example, the classes `B` and `C` in the linear hierarchy `A->B->C` uses virtual base classes. In this hierarchy, using virtual base classes is unnecessary and might cause confusion. Polyspace raises a violation. The class hierarchy `alpha->...->omega` is a diamond hierarchy and in this case, the use of virtual base classes is necessary. Polyspace does not flag this class hierarchy.

## Check Information

**Group:** Derived Classes
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M10-1-3

An accessible base class shall not be both virtual and non-virtual in the same hierarchy

## Description

### Rule Definition

*An accessible base class shall not be both virtual and non-virtual in the same hierarchy.*

### Rationale

The checker flags situations where the same class is inherited as a virtual base class and a non-virtual base class in the same derived class. These situations defeat the purpose of virtual inheritance and causes multiple copies of the base class sub-object in the derived class object.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Base Class Both Virtual and Non-Virtual in Same Hierarchy

```
class Base {};
class Intermediate1: virtual public Base {};
class Intermediate2: virtual public Base {};
class Intermediate3: public Base {};
class Final: public Intermediate1, Intermediate2, Intermediate3 {}; //Noncompliant
```

In this example, the class `Base` is inherited in `Final` both as a virtual and non-virtual base class. The `Final` object contains at least two copies of a `Base` sub-object.

## Check Information

**Group:** Derived Classes
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M10-2-1

All accessible entity names within a multiple inheritance hierarchy should be unique

## Description

### Rule Definition

*All accessible entity names within a multiple inheritance hierarchy should be unique.*

### Rationale

Data members and nonvirtual functions within the same inheritance hierarchy that have the same name might cause developer confusion. The entity the developer intended for use might not be the entity the compiler chooses. Avoid using nonunique names for accessible entities within a multiple inheritance hierarchy.

### Polyspace Implementation

This checker flags entities from separate classes that belong to the same derived class if they have an ambiguous name. The name of an entity is ambiguous if:

- Two variables share the same name, even if they are of different types.
- Two functions share the same name, same parameters, and the same return type.

If the data member accessed in the derived class is ambiguous, Polyspace reports this issue as a compilation issue, not a coding rule violation. The checker does not check for conflicts between entities of different kinds such as member functions against data members.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant Data Members in a Multiple Inheritance Hierarchy

```
#include <iostream>
#include <cstdlib>
#include <cstdint>

using namespace std;

class A {
  public:
    int32_t num;
    int32_t total;
    int32_t sum(int32_t toSum)
    {
        total = toSum + num;
        return total;
    };
};

class B {
  public:
    int32_t num;   //Also in class A
    int32_t total();
    int32_t sum(int32_t toSum)   //Also in class A
    {
        num = toSum + num;
        return num;
    };
};


class C : public A, public B { //Noncompliant
  public:
    void foo() {
        num = total;
        sum(num);
    }
};
```

- Because `class A` and `class B` define their own local variable `int32_t num`, and because `class C` is a multiple inheritance hierarchy containing `class A` and `class B`, Polyspace flags both `int32_t num` variables as noncompliant.
- Because `int32_t sum()` in `class A` and `int32_t sum()` in `class B` share the same name, return type, arguments, and are members of the same multiple inheritance hierarchy, both functions are flagged by Polyspace as noncompliant.
- Because `int32_t total` and `int_32t total()` are different types of class members, Polyspace does not flag them even though they are part of the same multiple inheritance hierarchy.

The ambiguous data members might be reported as compilation issues.

### Nonunique Entity Names When Classes Derive From Templates

Because templates generate code, a class hierarchy deriving from a template might have nonunique names. In this example, the classes `Deriv1` and `Deriv2` derives from two separate instances of the template `myObj`. Then, the class `FinalDerived` derives from both `Deriv1` and `Deriv2`. The template `myObj` generates two version of the function `get()` in the `FinalDerived` class: one from the `myObj<int>` instance and one from the `myObj<short>` instance. The nonunique name might result in unexpected behavior. Polyspace reports a violation

```
template <typename T>
class myObj {
    //...
public:
    void get(void);
};

class Deriv1: public myObj<int> {
public:
    //...
};

class Deriv2: public myObj<short> {
public:
    //...
};

class FinalDerived: public Deriv1, public Deriv2 { //Noncompliant

public:
    //...
};
```

### Check Information

**Group:** Derived Classes
**Category:** Advisory, Automated

## Version History

**Introduced in R2019a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M10-3-3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual

## Description

### Rule Definition

*A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.*

### Rationale

In C++, an abstract class is the base of a polymorphic class hierarchy and the derived classes implement variation of the abstract class. When a virtual function is overridden in a derived class by a pure virtual function, the derived class becomes an abstract class. That a derived class is defined as an abstract class or an implemented function is overridden by a pure virtual function is unexpected behavior, which might confuse a developer.

### Polyspace Implementation

Polyspace flags a pure virtual function if it overrides a function that is not pure virtual.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Do Not Redeclare Functions as Pure Virtual

```
class Conic{
    //...
    public:
    double centerAbscissa;
    double centerOrdinate;
    //..
    virtual double  getArea()=0;
};
class Circle: public Conic{
    //...
    public:
    //...
    double getArea() override{
        //calculate area of circle
    }
};
class Ellipse: public Circle{
    //...
    public:
    //...
    virtual double getArea()=0; //Noncompliant
};
```

In this example, the base class `Conic` is an abstract class because the function `getArea()` is a pure virtual function. The derived class `Circle` implements the function `getArea`. The expectation from such a polymorphic hierarchy is that the virtual function `getArea` calculates the area correctly based on the derived class. When the derived class `Ellipse` redeclares `getArea` as a pure virtual function, the derived class `Ellipse` becomes abstract and the function `Ellipse.getArea()` cannot be invoked. Developers might expect `Ellipse.getArea()` to return the area of the ellipse. Because this redeclaration as a pure virtual function does not meet developer expectation, Polyspace flags the declaration.

## Check Information

**Group:** Derived Classes
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M11-0-1

Member data in non-POD class types shall be private

## Description

### Rule Definition

*Member data in non-POD class types shall be private.*

### Rationale

If classes have data members that are publicly accessible, other classes and functions might interact with the class data members directly. Any change in the class might require updating the clients that use the class. If a class is not a plain-old-data (POD) type, restricting access to its data members enables encapsulation of the class. In such an encapsulated class, the implementation details of the class are opaque to the clients that use it. The class retains control over its implementation and can be maintained independently without impacting the clients that use the class.

### Polyspace Implementation

Polyspace flags nonprivate data members in classes that are not POD types. Polyspace space uses the same definition of POD classes as the standard.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Declare Data Members in Non-POD Classes as `private`

```
class nonPOD{
    nonPOD(){
        //...
    }
    ~nonPOD(){
        //...
    }
    public:
    int getX();
    int setX(int&);
    int getY();
    int setY(int&);
    int getZ();
    int setZ(int&);
    int x; //Noncompliant
    protected:
    int y; //Noncompliant
    private:
    int z;
};
```

In this example, the data members `y` and `z` are not `private`. Polyspace flags them.

## Check Information

**Group:** Member Access Control
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M12-1-1

An object's dynamic type shall not be used from the body of its constructor or destructor

## Description

### Rule Definition

*An object's dynamic type shall not be used from the body of its constructor or destructor.*

### Rationale

The dynamic type of an object is the type of its most derived class. For instance:

```
struct B {
    virtual ~B() {}
};
struct D: B {};
D d;
B* ptr = &d;
```

The dynamic type of the object pointed to by `*ptr` is `D` because that is the most derived class in the polymorphic hierarchy.

When you invoke the dynamic type of a polymorphic object in its constructor or destructor, you might get the type of the constructed or destroyed object instead of the type of the most derived object. This is because when you invoke the dynamic type during construction or destructor, the derived classes might not be constructed yet. Using dynamic types in constructors and destructors might result in unexpected behavior. Calling pure virtual functions from constructors and destructors results in undefined behavior. Avoid using the dynamic type of an object in its constructors or destructors.

### Polyspace Implementation

Polyspace flags these items when they are used in a constructor or a destructor of a polymorphic class:

- The operator `typeid`
- Virtual or pure virtual functions
- The function `dynamic_cast` or implicit C-style casts

Polyspace assumes that a class is polymorphic if it has any virtual member.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using Dynamic Type in Constructors and Destructors

```
#include <cassert>
#include <typeinfo>
```

```
class PS
{
public:
    PS ( )
    {
        typeid ( PS );                // Compliant
    }
};

class PS_1
{
public:
    virtual ~PS_1 ( );
    virtual void bar ( );
    PS_1 ( )
    {
        typeid ( PS_1 );                // Noncompliant
        PS_1::bar ( );                  // Compliant
        bar ( );                      // Noncompliant
        dynamic_cast< PS_1* > ( this ); // Noncompliant
    }
};
```

In this example, class PS has no virtual member. Polyspace does not consider PS a polymorphic class. Because PS is not polymorphic, its dynamic type does not change at run time. Polyspace does not flag using the `typeid` operator in the constructor `PS::PS()`.

PS_1 is considered polymorphic because it has a `virtual` member function. Because it is polymorphic, its dynamic type changes during run time. Polyspace flags the invocation of its dynamic type in the constructor `PS_1::PS_1()`.

## Check Information

**Group:** Special Member Functions
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule A14-5-1

A template constructor shall not participate in overload resolution for a single argument of the enclosing class type

## Description

### Rule Definition

*A template constructor shall not participate in overload resolution for a single argument of the enclosing class type.*

### Rationale

A template constructor can lead to confusion about which copy or move constructor is being invoked in a copy or move. For instance:

- An implicit constructor might be invoked when you expect the template constructor to be used. An implicit copy or move constructor exists in the class because a template constructor does not prevent its definition.
- The template constructor might be invoked when you expect an explicit constructor to be used. A template constructor might a better match than the explicit constructor when an overload is resolved.

### Polyspace Implementation

The checker raises a violation when:

- A class contains a template copy or move constructor but at least one copy or move uses the implicit constructor in the class.

  The violation is shown on the template constructor. Events associated with the result show the copy or move where an implicit constructor is invoked.
- A class contains a template copy or move constructor and an explicit constructor but at least one copy or move uses the template constructor.

  The violation is shown on the template constructor. Events associated with the result show the copy or move where the template constructor is invoked and the explicit constructor definition.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Implicit Constructor Used Instead of Template Constructor

```
class record
{
  public:
    template<typename T>
    record(const T &);  /*Non-compliant*/
};

void lookup(record aRecord)
{
    record copyOfARecord {aRecord};
}
```

In this example, when creating the object `copyOfARecord` from the object `aRecord`, a copy constructor must be invoked. The template constructor specialization, that is, `record<record>(const record &)`, and the implicit constructor, that is, `record(const record &)` are equally good candidates for a copy constructor. When a function and a specialization are equally good matches in the overload resolution process, the function is preferred. In this case, the implicit constructor is called instead of the template constructor, but a developer or reviewer might expect otherwise.

#### Template Constructor Used Instead of Explicit Constructor

```
class record
{
  public:
    record(const record &);

    template<typename T>
    record(T &);  /*Non-compliant*/
};

void lookup(record aRecord)
{
    record copyOfARecord {aRecord};
}
```

In this example, when creating the object `copyOfARecord` from the object `aRecord`, a copy constructor must be invoked. The template constructor specialization, that is, `record<record>(record &)` is a better match compared to the explicit constructor, that is, `record(const record &)`. The template constructor is called instead of the explicit constructor, but a developer or reviewer might expect otherwise.

## Check Information

**Group:** Templates
**Category:** Required, Automated

# Version History

**Introduced in R2021a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M14-5-3

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter

## Description

### Rule Definition

*A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.*

### Rationale

When declaring a user-defined assignment operator, the corresponding implicit operator is suppressed. When declaring a template assignment operator that has a generic parameter, this behavior is not preserved. In that case, to suppress the implicit shallow-copying operator, explicitly instantiate a version of the copy assignment operator for the class.

If you do not declare the copy assignment operator for the class, the compiler-generated copy assignment operator might be used instead on implementation. Not declaring a copy assignment operator explicitly might result in an unexpected outcome, such as creating a shallow copy when a deep copy was intended.

### Polyspace Implementation

Polyspace flags this checker if a structure, class, or union contains a template assignment operator that has a generic parameter but no copy assignment operator is present within the structure, class, or union.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Noncompliant Template Assignment Operator That Has Generic Parameter

```
#include<cstdint>
namespace example
{
  class A          // Noncompliant
  {
    public:

      template <typename T>
      T & operator= ( T const & rhs )
      {
        if ( this != &rhs ) {
          delete i;
          i = new int32_t;
          *i = *rhs.i;
        }
        return *this;
      }
    private:
      int32_t * i;      // Member requires deep copy
  };

  void f ( A const & a1, A & a2 )
  {
    a2 = a1;
  }
};
```

Because no copy assignment operator is declared within the class, Polyspace flags `class A`. The implicitly defined copy assignment operator is not suppressed by the template assignment operator and results in a shallow copy of a1 to a2 when you might want a deep copy.

### Template Assignment Operator That Has a Generic Parameter and Copy Assignment Operator Declared

```
#include<cstdint>
namespace example
{
  class A
  {
    public:
      A & operator= (A const & rhs) {};    //Compliant

      template <typename T>
      T & operator= ( T const & rhs )      //Compliant
      {
        if ( this != &rhs ) {
          delete i;
          i = new int32_t;
          *i = *rhs.i;
        }
        return *this;
      }
    private:
      int32_t * i;
  };

  void f ( A const & a1, A & a2 )
  {
    a2 = a1;
  }
};
```

Because this class contains a copy assignment operation declaration, Polyspace does not flag `class A`.

## Check Information

**Group:** Templates
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M14-6-1

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->

## Description

### Rule Definition

*In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.*

### Rationale

When a class template derives from another class template, there might be confusion arising from the use of names that exist in both the base template and the current scope or namespace. When the same name exists in the base class template and a namespace that contains the classes, the scope resolution of these names is dependent on the compiler, which might be contrary to developer's expectation. To avoid confusion, use fully qualified id or `this->` to explicitly disambiguate the intended object when such a name conflict exists.

### Polyspace Implementation

Polyspace flags names for which all of these conditions are true:

- The name exists in the base class.
- The name exists in a namespace that contains the base class.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Use Fully Qualified Names in Class Templates That Have Dependent Base Classes

```
typedef signed   int          int32_t;
namespace NS0{
    typedef int32_t TYPE;

    void bar( );
    namespace NS1{
        namespace NS{

            template <typename T>
            class Base;
            template <typename T>
            class Derived : public Base<T>
            {
                void foo ( )
                {
                    TYPE t = 0;                           // Noncompliant
                    bar ( );                                // Noncompliant
                }
                void foo2 ( )
                {
                    NS0::TYPE t1 = 0;                       // Compliant
                    NS0::bar ( );                             // Compliant
                    typename Base<T>::TYPE t2 = 0; // Compliant
                    this->bar ( );                // Compliant
                }
            };
            template <typename T>
            class Base
            {
            public:
                typedef T TYPE;
                void bar ( );
            };
            template class Derived<int32_t>;
        }
    }
}
```

In this example, the names Type and bar are defined both in the namespace NS0 and within the class template Base. The class template Derived derives from Base. In Derived::foo1(), these names are used without using the fully qualified names or this->. It is not clear whether the TYPE in Base::foo1 resolves to NS0::TYPE or Base::TYPE. You might get different results depending on the implementation of the compiler. Polyspace flags these ambiguous statements.

In Derived::foo2(), TYPE and bar are invoked by using their fully qualified name or this->. By using qualified names or this->, the ambiguity in scope resolution is bypassed. Polyspace does not flag these uses.

## Check Information

**Group:** Templates
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M15-0-3

Control shall not be transferred into a try or catch block using a goto or a switch statement

## Description

### Rule Definition

*Control shall not be transferred into a try or catch block using a goto or a switch statement.*

### Rationale

Transferring control into a `try` or `catch` block by using a `goto` or a `switch` statement results in ill-formed code that is difficult to understand. The intended behavior of such code is difficult to identify and the code might result in unexpected behavior. Abruptly entering into an exception handling block might cause compilation failure in some compilers while other compilers might not diagnose the issue. To improve code understanding and reduce unexpected behavior, avoid transferring control into a try or a catch block.

### Polyspace Implementation

Polyspace flags the `goto` and `switch` statements that jump into a `try` or a `catch` block.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Jumping into `try` or `catch` Blocks

```
#include<cstdint>
void foo ( int32_t input )
{
    if ( input==1 )
    {
        goto Label_1; // Noncompliant
    }
    if ( input==2 )
    {
        goto Label_2; // Noncompliant
    }
    switch ( input ) //Noncompliant
    {
    case 1:
        try
        {
            Label_1:
        case 2:
            break;
        }
        catch ( ... )
        {
            Label_2:
        case 3:
            break;
        }
        break;
    default:
        {
            //...
            break;
        }
    }
}
```

In this example, `goto` and `switch` statements are used to jump into a `try-catch` block. Jumping into a try-catch block makes the code difficult to understand. Abrupt transfer of control into a `try` block or a `catch` block might result in compilation failure. Polyspace flags the `goto` and `switch` statements. Such transfer of control into `try-catch` blocks might cause compilation failures.

## Check Information

**Group:** Exception Handling
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M15-1-1

The assignment-expression of a throw statement shall not itself cause an exception to be thrown

## Description

### Rule Definition

*The assignment-expression of a throw statement shall not itself cause an exception to be thrown.*

### Rationale

In C++, you can use a `throw` statement to raise exceptions explicitly. The compiler executes such a `throw` statement in two steps:

- First, it creates the argument for the `throw` statement. The compiler might call a constructor or evaluate an assignment expression to create the argument object.
- Then, it raises the created object as an exception. The compiler tries to match the exception object to a compatible handler.

If an unexpected exception is raised when the compiler is creating the expected exception in a `throw` statement, the unexpected exception is raised instead of the expected one. Consider this code where a `throw` statement raises an explicit exception of class `myException`.

```
class myException{
    myException(){
        msg = new char[10];
        //...
    }
    //...
};

foo(){
    try{
        //..
        throw myException();
    }
    catch(myException& e){
        //...
    }
}
```

During construction of the temporary `myException` object, the `new` operator can raise a `bad_alloc` exception. In such a case, the `throw` statement raises a `bad_alloc` exception instead of `myException`. Because `myException` was the expected exception, the catch block is incompatible with `bad_alloc`. The `bad_alloc` exception becomes an unhandled exception. It might cause the program to abort abnormally without unwinding the stack, leading to resource leak and security vulnerabilities.

Unexpected exceptions arising from the argument of a `throw` statement can cause resource leaks and security vulnerabilities. To prevent such unwanted outcome, avoid using expressions that might raise exceptions as argument in a `throw` statement.

### Polyspace Implementation

Polyspace flags the expressions in `throw` statements that can raise an exception. Expressions that can raise exceptions can include:

- Functions that are specified as `noexcept(false)`
- Functions that contain one or more explicit `throw` statements
- Constructors that perform memory allocation operations
- Expressions that involve dynamic casting

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Expressions That Can Raise Exceptions in `throw` Statements

This example shows how Polyspace flags the expressions in `throw` statements that can raise unexpected exceptions.

```
int f_throw() noexcept(false);

class WithDynamicAlloc {
public:
    WithDynamicAlloc(int n) {
        m_data = new int[n];
    }
    ~WithDynamicAlloc() {
        delete[] m_data;
    }
private:
    int* m_data;
};

class MightThrow {
public:
    MightThrow(bool b) {
        if (b) {
            throw 42;
        }
    }
};

class Base {
    virtual void bar() =0;
};
class Derived: public Base {
    void bar();
};
class UsingDerived {
public:
    UsingDerived(const Base& b) {
        m_d =
        dynamic_cast<const Derived&>(b);
    }
private:
    Derived m_d;
};
class CopyThrows {
public:
    CopyThrows() noexcept(true);
    CopyThrows(const CopyThrows& other) noexcept(false);
};
int foo(){
    try{
        //...
        throw WithDynamicAlloc(10); //Noncompliant
        //...
        throw MightThrow(false);//Noncompliant
        throw MightThrow(true);//Noncompliant
        //...
        Derived d;
        throw  UsingDerived(d);// Noncompliant
        //...
        throw f_throw(); //Noncompliant
        CopyThrows except;
        throw except;//Noncompliant
    }
    catch(WithDynamicAlloc& e){
        //...
    }
    catch(MightThrow& e){
        //...
    }
    catch(UsingDerived& e){
        //...
    }
}
```

- When constructing a `WithDyamicAlloc` object by calling the constructor `WithDynamicAlloc(10)`, exceptions can be raised during dynamic memory allocation. Because the expression `WithDynamicAlloc(10)` can raise an exception, Polyspace flags the `throw` statement `throw WithDynamicAlloc(10);`
- When constructing a `UsingDerived` object by calling the constructor `UsingDervide()`, exceptions can be raised during the dynamic casting operation. Because the expression `UsingDerived(d)` can raise exceptions, Polyspace flags the statement `throw UsingDerived(d)`.
- In the function `MightThrow()`, exceptions can be raised depending on the input to the function. Because Polyspace analyzes functions statically, it assumes that the function `MightThrow()` can raise exceptions. Polyspace flags the statements `throw MightThrow(false)` and `throw MightThrow(true)`.
- In the statement `throw except`, the object `except` is copied by implicitly calling the copy constructor of the class `CopyThrows`. Because the copy constructor is specified as `noexcept(false)`, Polyspace assumes that the copy operation might raise exceptions. Polyspace flags the statement `throw except`.
- Because the function `f_throw()` is specified as `noexcept(false)`, Polyspace assumes that it can raise exceptions. Polyspace flags the statement `throw f_throw()`.

## Check Information

**Group:** Exception handling
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M15-1-2

NULL shall not be thrown explicitly

## Description

### Rule Definition

*NULL shall not be thrown explicitly.*

### Rationale

The macro `NULL` is commonly used to refer to null pointers. Compliers interpret `NULL` as an integer with value zero, instead of a pointer. When you use `NULL` explicitly in a `throw` statement, you might expect the statement to raise a pointer type exception. The `throw(NULL)` is equivalent to `throw(0)` and raises an integer exception. This behavior might be contrary to developer expectation and might result in bugs that are difficult to find. Avoid using `NULL` explicitly in a `throw` statement.

### Polyspace Implementation

Polyspace flags a `throw` statement that raises a `NULL` explicitly. Polyspace does not flag the statement when `NULL` is raised after casting to a specific type or assigning it to a pointer type.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Do Not Throw `NULL` Explicitly

```
typedef          char         char_t;
typedef signed   int          int32_t;

#include <cstddef>

void foo()
{
    try {
        char_t * p1 = NULL;
        throw ( NULL );            // Noncompliant
        throw(p1); //Compliant
        throw ( static_cast < const char_t * > ( NULL ) ); // Compliant
    } catch ( int32_t i ) {        // NULL exception handled here
        // /*...*/
    } catch ( const char_t * ) { // Other two exceptions are handled here
        // /*...*/
    }
}
```

In this example, three exceptions are raised directly by using `throw` statements.

- Polyspace flags the statement `throw(NULL)` because it explicitly raises `NULL` as exception. You might expect that this statement raises a pointer type exception that is handled in the second `catch` block. This statement actually raises an `int` exception that is handled in the first `catch` block.
- The other `throw` statements show the compliant method of using `NULL` in a `throw` statement. For instance, the second `throw` statement raises a `char*` that is assigned the value `NULL`. The third `throw` statement raises a `char*` by casting `NULL` to a `char*`. Because these statements do not raise `NULL` explicitly, Polyspace does not flag them.

## Check Information

**Group:** Exception Handling
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M15-1-3

An empty throw (throw;) shall only be used in the compound statement of a catch handler

## Description

### Rule Definition

*An empty throw (throw;) shall only be used in the compound statement of a catch handler.*

### Rationale

When you use an empty throw statement `(throw;)`, the compiler checks if an exception object is present in the current scope. If the current scope contains an exception object, the compiler raises a temporary object containing the current exception. If the current scope does not contain an exception objects, the compiler invokes `std::terminate()` implicitly. The function `std::terminate()` terminates the program execution in an implementation-defined manner. That is, the exact process of program termination depends on the software and hardware that you are using. For instance, `std:terminate()` might invoke `std::abort()` to abnormally abort the execution without unwinding the stack, leading to resource leak and security vulnerabilities.

The best practice is to use an empty throw statement only in the `catch` block of a `try-catch` construct, which enables you to spread the handling of an exception across multiple catch blocks. Avoid using empty throw statements in scopes that might not contain an exception.

### Polyspace Implementation

Polyspace flags an empty throw statement if it is not within a catch block.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using Empty `throw` Statements Outside `catch` Blocks

```
#include <iostream>
#include <typeinfo>
#include <exception>

void handleException()//function to handle all exception
{
    try {
        throw; // Noncompliant
    }
    catch (std::bad_cast& e) {
        //...Handle bad_cast...
    }
    catch (std::bad_alloc& e) {
        //...Handle bad_alloc...
    }
    catch(...){
        //...Handle other exceptions
    }
}

void f()
{
    try {
        //...something that might throw...
    }
    catch (...) {
        handleException();
    }
}
```

In this example, the function `handleException()` raises the current exception by using an empty throw statement, and then directs it to the appropriate `catch` block. This method of delegating the exception handling works as intended only when the function `handleException()` is called from within a `catch` block. The empty throw statement might cause abrupt termination of the program if the function is called in any other scope that does not contain an exception. Polyspace flags the empty throw statement.

#### Use Empty `throw` Statement to Handle Exceptions in Multiple Blocks

```
#include <iostream>
#include <typeinfo>
#include <exception>
void foo()//function to handle all exception
{
    try {
        //...
    }
    catch (std::bad_cast& e) {
        //...Handle bad_cast...
    }
    catch (std::bad_alloc& e) {
        //...Handle bad_alloc...
    }
    catch(std::exception& e){
        //...Handle std::exceptions
        // if exception cannot be handled
        // throw it again
        throw;//Compliant
    }
}

int main(){
    try{
        foo();
    }
    catch(...){

    }
}
```

This example shows a compliant use of an empty throw statement. The function `foo` contains a `try-catch` construct that handles specific exceptions. If the raised exception cannot be handled, `foo` raises the exception again as an unhandled exception by using an empty throw statement. In `main`, the function `foo` is invoked and any unhandled exception arising from `foo` is handled in a generic `catch(...)` block. By using the empty throw statement, the handling of the exception is spread across the catch blocks of `foo` and `main`. In this case, the empty throw statement is executed only when there is an exception in the same scope because it is within a `catch` block. Polyspace does not flag it.

## Check Information

**Group:** Exception Handling
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M15-3-1

Exceptions shall be raised only after start-up and before termination

## Description

### Rule Definition

*Exceptions shall be raised only after start-up and before termination.*

### Rationale

In C++, the process of exception handling runs during execution of `main()`, where exceptions arising in different scopes are handled by exception handlers in the same or adjacent scopes. Before starting the execution of `main()`, the compiler is in startup phase, and after finishing the execution of `main()`, the compiler is in termination phase. During these two phases, the compiler performs a set of predefined operations but does not execute any code.

If an exception is raised during either the startup phase or the termination phase, you cannot write an exception handler that the compiler can execute in those phases. For instance, you might implement `main()` as a `function-try-catch` block to handle exceptions. The `catch` blocks in `main()` can handle only the exceptions raised in `main()`. None of the `catch` blocks can handle exceptions raised during startup or termination phase. When such exceptions are raised, the compiler might abnormally terminate the code execution without unwinding the stack. Consider this code where the construction and destruction of the static object `obj` might cause an exception.

```
class A{
    A(){throw(0);}
    ~A(){throw(0)}
};

static A obj;

main(){
    //...
}
```

The static object `obj` is constructed by calling `A()` before `main()` starts, and it is destroyed by calling `~A()` after `main()` ends. When `A()` or `~A()` raises an exception, an exception handler cannot be matched with them. Based on the implementation, such an exception can result in program termination without stack unwinding, leading to memory leak and security vulnerabilities.

Avoid operations that might raise an exception in the parts of your code that might be executed before startup or after termination of the program. For instance, avoid operations that might raise exceptions in the constructor and destructor of static or global objects.

### Polyspace Implementation

Polyspace flags a global or a a static variable declaration that uses a callable entity that might raise an exception. For instance:

- Function: When you call an initializer function or constructor directly to initialize a global or static variable, Polyspace checks whether the function raises an exception and flags the variable declaration if the function might raise an exception. Polyspace deduces whether a function might raise an exception regardless of its exception specification. For instance, if a `noexcept` constructor raises an exception, Polyspace flags it. If the initializer or constructor calls another function, Polyspace assumes the called function might raise an exception only if it is specified as `noexcept(<false>)`. Some standard library functions, such as the constructor of `std::string`, use pointers to functions to perform memory allocation, which might raise exceptions. Polyspace does not flag the variable declaration when these functions are used.

- External function: When you call external functions to initialize a global or static variable, Polyspace flags the declaration if the external function is specified as `noexcept(<false>)`.
- Virtual function: When you call a virtual function to initialize a global or static variable, Polyspace flags it if the virtual function is specified as `noexcept(<false>)` in any derived class. For instance, if you use a virtual initializer function that is declared as `noexcept(<true>)` in the base class, and `noexcept(<false>)` in a subsequent derived class, Polyspace flags it.
- Pointers to function: When you use a pointer to a function to initialize a global or static variable, Polyspace assumes that pointer to a function do not raise exceptions.

Polyspace ignores:

- Exceptions raised in destructors
- Exceptions raised in `atexit()` operations

Polyspace also ignores the dynamic context when checking for exceptions. For instance, you might initialize a global or static variable by using a function that raises exceptions only in a certain dynamic context. Polyspace flags such a declaration even if the exception might never be raised. You can justify such a violation by using comments in Polyspace.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Exceptions Before `main()` Starts

This example shows how Polyspace flags construction or initialization of a global or static variable that might raise an exception. Consider this code where static and global objects are initialized by using various callable entities.

```
#include <stdexcept>
#include <string>
class C
{
public:
    C ( ){throw ( 0 );}
    ~C ( ){throw ( 0 );}
};
int LibraryFunc();
int LibraryFunc_noexcept_false() noexcept(false);
int LibraryFunc_noexcept_true() noexcept(true);
int  g() noexcept {
    throw std::runtime_error("dead code");
    return 0;
}
int f() noexcept {
    return g();
}
int init(int a) {
    if (a>10) {
        throw std::runtime_error("invalid case");
    }
    return a;
}
void* alloc(size_t s) noexcept {
    return new int[s];
}
int a = LibraryFunc() +
LibraryFunc_noexcept_true();            // Compliant
int global_int =
LibraryFunc_noexcept_false() +          // Noncompliant
LibraryFunc_noexcept_true();
static C static_c;                      //Noncompliant
static C static_d;                      //Compliant
C &get_static_c(){
    return static_c;
}
C global_c;                             //Noncompliant
int a3 = f();                           //Compliant
int b3 = g();                           //Noncompliant
int a4 = init(5);                       //Noncompliant
int b5 = init(20);                      //Noncompliant
int* arr = (int*)alloc(5);              //Noncompliant

int main(){
    //...
}
```

- The global pointer `arr` is initialized by using the function `alloc()`. Because `alloc()` uses `new` to allocate memory, it can raise an exception when initializing `arr` during the startup of the program. Polyspace flags the declaration of `arr` and highlights the use of `new` in the function `alloc()`.
- The integer variable `b3` is initialized by calling the function `g()`, which is specified as `noexcept`. Polyspace deduces that the correct exception specification of `g()` is `noexcept(false)` because it contains a `throw()` statement. Initializing the global variable `b3` by using `g()` might raise an exception when initializing `arr` during the startup of the program. Polyspace flags the declaration of `b3` and highlights the `throw` statement in `g()`. The declaration of `a3` by calling `f()` is not flagged. Because `f()` is a `noexcept` function that does not throw, and calls another `noexcept` function, Polyspace deduces that `f()` does not raise an exception.
- The global variables `a4` and `b5` are initialized by calling the function `init()`. The function `init()` might raise an exception in certain cases, depending on the context. Because Polyspace deduces the exception specification of a function statically, it assumes that `init()` might raise an exception regardless of context. Consequently, Polyspace flags the declarations of both `a4` and `b5`, even though `init()` raises an exception only when initializing `b5`.
- The global variable `global_int` is initialized by calling two external functions. The external function `LibraryFunc_noexcept_false()` is specified as `noexcept(false)` and Polyspace assumes that this external function might raise an exception. Polyspace flags the declaration of `global_int`. Polyspace does not flag the declaration of `a` because it is initialized by calling external functions that are not specified as `noexcept(false)`.

- The static variable `static_c` and the nonstatic global variable `global_c`is declared and initialized by using the constructor of the class `C`, which might raise an exception. Polyspace flags the declarations of these variables and highlights the `throw()` statement in the constructor of class `C`. Polyspace does not flag the declaration of the unused static variable `static_d`, even though its constructor might raise an exception. Because it is unused, `static_d` is not initialized and its constructor is not called. Its declaration does not raise any exception.

## Check Information

**Group:** Exception Handling
**Category:** Required, Automated

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M15-3-3

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases

## Description

### Rule Definition

*Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.*

### Rationale

The handler `catch` blocks of a function `try` block handle exception that are raised from the body of the function and the initializer list. When used in class constructors and destructors, these `catch` blocks might handle exceptions that arise during the creation or destruction of the class nonstatic members. That is, the `catch` blocks might be executed before or after the lifetime of the nonstatic members of a class. If the nonstatic members of a class are accessed in such `catch` blocks, the compiler might attempt to access objects that are not created yet or already deleted, which is undefined behavior. For instance:

```
class C{

    private:
    int* inptr_x;
    public:
    C() try: inptr_x(new int){}
    catch(...){
        intptr_x = nullptr;
        //...
    }
};
```

Here, the constructor of `C` is implemented by using a function `try` block to handle any exception arising from the memory allocation operation in the initializer list. In the `catch` block of this function-`try` block, the class member `C.intptr_x` is accessed. The catch block executes when the memory allocation for `intptr_x` failed. That is, the catch block attempts to access the member before its lifetime, which is undefined behavior.

To avoid undefined behavior, avoid using the nonstatic data members or base classes of an object in the catch block of the function-try-block implementation of its constructors and destructor.

### Polyspace Implementation

If a statement in the catch block of a constructor or destructor function-`try` block accesses any of these, Polyspace flags the statement:

- The nonstatic members of the object
- The base classes of the object
- The nonstatic members of the base classes

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Accessing Nonstatic Members of Classes in `function-try` `catch` Blocks

```
#include<cstdint>
class B
{
public:
    B ( ) try: x(0){/*...*/}
    catch ( ... )
    {
        if ( 0 == x ){/*...*/}  //Noncompliant
        //...
    }
    ~B ( ) try{/*...*/}
    catch ( ... )
    {
        if ( 0 == x ){/*...*/} //Noncompliant
        //...
        else if (sb == 1){/*...*/} //Compliant
        //....
    }
public:
    static int32_t sb;
protected:
    int32_t x;
};

class D : public B
{
public:
    D ( ) try: B(),y{0}{/*...*/}
    catch ( ... )
    {
        if ( 0 == x ){/*...*/}  //Noncompliant
        //...
        else if (y == 1){/*...*/}   //Noncompliant
        //...
    }
    ~D ( )try {/*...*/}
    catch ( ... )
    {
        if ( 0 == x ) {/*...*/} //Noncompliant
        //...
    }
protected:
    int32_t y;
};
```

In this example, the constructors and destructors of `B` and `D` are implemented by using function-try blocks. The `catch` blocks of these function-`try` blocks access the nonstatic members of the class and its base class. Polyspace flags accessing these nonstatic members in the `catch` blocks. Because the lifetime of `static` members is greater than the lifetime of the object itself, Polyspace does not flag accessing `static` objects in these `catch` blocks.

## Check Information

**Group:** Exception Handling
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M15-3-4

Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point

## Description

### Rule Definition

*Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.*

### Rationale

In C++, when an operation raises an exception, the compiler tries to match the exception with a compatible exception handler in the current and adjacent scopes. If no compatible exception handler for a raised exception exists, the compiler invokes the function `std::terminate()` implicitly. The function `std::terminate()` terminates the program execution in an implementation-defined manner. That is, the exact process of program termination depends on the particular set of software and hardware that you use. For instance, `std::terminate()` might invoke `std::abort()` to abnormally abort the execution without unwinding the stack. If the stack is not unwound before program termination, then the destructors of the variables in the stack are not invoked, leading to resource leak and security vulnerabilities.

Consider this code where multiple exceptions are raised in the try block of code.

```
class General{/*...  */};
class Specific : public General{/*...*/};
class Different{}
void foo() noexcept
{
    try{
        //...
        throw(General e);
        //..
        throw( Specific e);
        // ...
        throw(Different e);
    }
    catch (General& b){

    }
}
```

The catch block of code accepts references to the base class `General`. This catch block is compatible with exceptions of the base class `General` and the derived class `Specific`. The exception of class `Different` does not have a compatible handler. This unhandled exception violates this rule and might result in resource leaks and security vulnerabilities.

Because unhandled exceptions can lead to resource leak and security vulnerabilities, match the explicitly raised exceptions in your code with a compatible handler.

### Polyspace Implementation

- Polyspace flags a `throw` statement in a function if a compatible catch statement is absent in the call path of the function. If the function is not specified as `noexcept`, Polyspace ignores it if its call path lacks an entry point like `main()`.
- Polyspace flags a `throw` statement that uses a `catch(…)` statement to handle the raised exceptions.
- Polyspace does not flag rethrow statements, that is, `throw` statements within catch blocks.
- You might have compatible catch blocks for the `throw` statements in your function in a nested try-catch block Polyspace ignores nested try-catch blocks. Justify `throw` statements that have compatible catch blocks in a nested structure by using comments. Alternatively, use a single level of try-catch in your functions.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Match `throw` Statements with Compatible Catch Blocks

This example shows how Polyspace flags operations that raise exceptions without any compatible handler. Consider this code.

```
#include <stdexcept>

class MyException : public std::runtime_error {
public:
    MyException() : std::runtime_error("MyException") {}
};

void ThrowingFunc() {
    throw MyException(); //Noncompliant
}

void CompliantCaller() {
    try {
        ThrowingFunc();
    } catch (std::exception& e) {
        /* ... */
    }
}

void NoncompliantCaller() {
    ThrowingFunc();
}

int main(void) {
    CompliantCaller();
    NoncompliantCaller();
}

void GenericHandler() {
    try {
        throw MyException(); //Noncompliant
    } catch (...) {
        /* ... */
    }
}

void TrueNoexcept() noexcept {
    try {
        throw MyException();//Compliant
    } catch (std::exception& e) {
        /* ... */
    }
}

void NotNoexcept() noexcept {
    try {
        throw MyException(); //Noncompliant
    } catch (std::logic_error& e) {
        /* ... */
    }
}
```

- The function `ThrowingFunc()` raises an exception. This function has multiple call paths:

  - `main()->CompliantCaller()->ThrowingFunc()`: In this call path, the function `CompliantCaller()` has a catch block that is compatible with the exception raised by `ThrowingFunc()`. This call path is compliant with the rule.
  - `main()->NoncompliantCaller()->ThrowingFunc()`: In this call path, there are no compatible handlers for the exception raised by `ThrowingFunc()`. Polyspace flags the `throw` statement in `ThrowingFunc()` and highlights the call path in the code.

  The function `main()` is the entry point for both of these call paths. If `main()` is commented out, Polyspace ignores both of these call paths. If you want to analyze a call path that lacks an entry point, specify the top most calling function as `noexcept`.
- The function `GenericHandler()` raises an exception by using a `throw` statement and handles the raised exception by using a generic catch-all block. Because Polyspace considers such catch-all handler to be incompatible with exceptions that are raised by explicit `throw` statements, Polyspace flags the `throw` statement in `GenericHandler()`.
- The `noexcept` function `TrueNoexcept()` contains an explicit `throw`statement and a catch block of compatible type. Because this `throw` statement is matched with a compatible `catch` block, it is compliant with the rule.
- The `noexcept` function `NotNoexcept()` contains an explicit `throw` statement, but the catch block is not compatible with the raised exception. Because this `throw` statement is not matched with a compatible catch block, Polyspace flags the `throw` statement in `NotNoexcept()`.

## Check Information

**Group:** Exception handling
**Category:** Required, Automated

# Version History

**Introduced in R2020b**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M15-3-6

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class

## Description

### Rule Definition

*Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.*

### Rationale

In a `try-catch` or `function-try` block, exception objects of a derived class match to handler `catch` blocks that accept the base class. If you place handlers of the base exception class before handlers of the derived exception class, the base class handler handles both base and derived class exceptions. The derived class handler becomes unreachable code, which is unexpected behavior. When using a class hierarchy to raise exceptions, make sure that the handler of a derived class precedes the handler of a base class.

### Polyspace Implementation

Polyspace flags a handler block if it follows a handler of a base class.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Order Handler Blocks from most Derived to Base Class

```
#include<exception>
// classes used for exception handling
class MathError { };
class NotANumber: public MathError { };
class DivideByZero: public NotANumber{};

void bar(void){
    try
    {
        // ...
    }
    catch ( MathError &e )
    {
        // ...
    }
    catch ( NotANumber &nan ) // Noncompliant
    {
        // Unreachable Code
    }
    catch (DivideByZero &dbz)//Noncompliant
    {
        //Unreachable Code
    }
}
```

In this example, three classes in a hierarchy might arise in the `try` block. The handler `catch` blocks handle the exceptions.

- The block `catch ( NotANumber &nan )` follows the handler of its base class `catch ( MathError &e )`. Because the exception of class `NotANumber` also matches to the handler `catch ( MathError &e )`, the handler block `catch ( NotANumber &nan )` becomes unreachable code. The order of this block is noncompliant with this rule. Polyspace flags the handler block.
- The block `catch ( DivideByZero &dbz )` becomes unreachable code because exceptions of the class `DivideByZero` match to the preceding handlers of its base classes. Polyspace flags the handler block `catch ( DivideByZero &dbz )`.

## Check Information

**Group:** Exception Handling
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M15-3-7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last

## Description

### Rule Definition

*Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.*

### Rationale

In a try-catch statement or function-try block, the compiler matches the raised exception with a `catch()` handler. The `catch(…)` handler matches any exception. Handlers after the catch-all handler within the same try-catch statement or function try-block are ignored by the compiler during the exception handling process and are unreachable code.

Having a handler after the catch-all handler might result in developer confusion as to why certain intended handlers are not being executed. Likewise, the catch-all handler might not handle the exception in the way the developer intends, resulting in confusion.

### Polyspace Implementation

Polyspace raises this defect whenever a handler appears after the catch-all handler within the try-catch statement or function try-block.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Handlers After the Catch-All Handler Are Noncompliant

```
#include <iostream>
#include <exception>

using namespace std;

int main() {

    try {
        // some code
    } catch (exception& e1) { // Compliant

        //...

    } catch (...) { // Compliant

        //...
    } catch (exception& e2) { // Noncompliant

        //...
    }

    return 0;
}
```

Because the `catch (exception& e2)` handler comes after the `catch(…)` handler, Polyspace flags the handler before the catch-all handler as noncompliant. This issue might cause a compilation error.

## Check Information

**Group:** Exception Handling
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M16-0-1

#include directives in a file shall only be preceded by other preprocessor directives or comments

## Description

### Rule Definition

*#include directives in a file shall only be preceded by other preprocessor directives or comments.*

### Rationale

Grouping all #include preprocessor directives at the beginning of the file makes the code more readable. #include directives might include header files where macros are defined. If you use such a macro before including its definition, you might encounter unexpected code behavior.

### Polyspace Implementation

Polyspace raises this defect when an `#include` directive comes after any code that is not a comment or preprocessor directive. Polyspace does not report a violation of this rule when an `#include` directive is located within an `extern "C"` block.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### #include Directive Preceded by Noncompliant Code

```
//this comment is compliant     //Compliant
/*
    This comment is compliant
*/

#ifndef TESTING_H               //Compliant
#define TESTING_H               //Compliant

#include <iostream>             //Compliant
using namespace std;            //Compliant
#include <exception>            //Noncompliant

#endif
```

Because an include directive follows a code statement that is neither a preprocessor directive nor a comment, `Polyspace` flags the include directive.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M16-0-2

Macros shall only be #define'd or #undef'd in the global namespace

## Description

### Rule Definition

*Macros shall only be #define'd or #undef'd in the global namespace.*

### Rationale

If you define or undefine macros in a local namespace, you might expect the macro to be valid only in the local namespace. But macros do not follow the scoping mechanism. Instead, the compiler replaces all occurrences of a macro by its defined value beginning at the `#define` statement until the end of file or until the macro is redefined. This behavior of macros might be contrary to developer expectation and might cause logic errors that result in bugs.

### Polyspace Implementation

Polyspace flags a `#define` or `#undef` statement that is placed within a block instead of in the global namespace.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Macros in Local Namespaces

```
#include<cstdlib>
#define HCUT 1
namespace unnormalized{
    #define HCUT 6582 //Noncompliant
    void foo(){
        //...
    }
};
void bar(){
    int intEnergy = HCUT*10;
    //HCUT is 6582, you might expect HCUT=1;
}

namespace uniteV{
    const double hcut = 6582; //eV
    void foo(){

    }
};
```

In this example, different values of `HCUT` are defined, perhaps to accommodate code written by using different systems of unit. You might expect the definition of `HCUT` in the namespace `unnormalized` to remain limited to the namespace. But the value of HCUT remains 6582 until the end of file. For instance, in the function bar, you might expect that HCUT is one, but the value of HCUT remains 6582, which might cause logic error, unexpected results, and bugs. Polyspace flags the #define statement within the local namespace.

To implement constants that might have different values in different scopes, use const variables, as shown in the namespace uniteV. Avoid using macros to represent constants that might require different values in different scopes.

### Check Information

**Group:** Preprocessing Directives
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

### See Also

Check AUTOSAR C++ 14 (-autosar-cpp14)

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M16-0-5

Arguments to a function-like macro shall not contain tokens that look like pre-processing directives

## Description

### Rule Definition

*Arguments to a function-like macro shall not contain tokens that look like pre-processing directives.*

### Rationale

When a compiler encounters function-like macros, it replaces the argument of the macro into the replacement code. If the argument contains a token that looks like preprocessing directives, the replacement process during macro expansion is undefined. Depending on the environment, such a function-like macro might behave in unexpected ways, leading to errors and bugs.

### Polyspace Implementation

Polyspace flags calls to function-like macros if their argument starts with the character #.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Avoid Arguments That Start with # in Function-Like Macros

```
#include<cstdlib>
#include<iostream>
#define PRINT(ARG) std::cout<<#ARG
//....
#define Error1
//...


void foo(void){
    PRINT(
    #ifdef Error1  //Noncompliant
    "Error 1"
    #else    //Noncompliant
    "Error 2"
    #endif
    );

}
```

In this example, the function-like macro `PRINT` is invoked with an argument that chooses between two strings by using an `#ifdef` block. Depending on the environment, the output of this code might be `#ifdef Error1 //Noncompliant "Error 1" #else "Error 2" #endif // Noncompliant` or `Error 1`. Polyspace flags the arguments that start with the character #.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M16-0-6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##

## Description

### Rule Definition

*In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.*

### Rationale

When you invoke function-like macros, the compiler expands the macro by replacing its parameters with the tokens. Then the compiler substitutes the expanded macro into the code. This expansion and substitution process does not take precedence of operation into account. The function-like macros might produce unexpected results if their parameters are not enclosed in parenthesis. For instance, consider this function-like macro:

```
#define dustance_from_ten(x) x>10? x-10:10-x
```

The macro is intended to measure the distance of a number from ten. When you invoke the macro with the argument (a-b), the macro expands to:

```
a-b>10: a-b-10:10-a-b
```

The expression `10-a-b` is equivalent to `10-(a+b)` instead of the intended distance `10-(a-b)`. This unexpected behavior might result in errors and bugs. To avoid such unexpected behaviors, enclose parameters of a function-like macro in parentheses.

The exception to this rule is when a parameter is used as an operand of # or ##.

### Polyspace Implementation

Polyspace flags function-like macro definitions if the parameters are not enclosed in parenthesis. Polyspace does not flag unparenthesized parameters if they are preceded by the operators `.`, `->`, or the characters #, ##.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Enclose Parameters of Function-Like Macros in Parentheses

```
#include<iostream>
#include<cmath>
#define abs(x) (x>0) ? x:-x //Noncompliant

double foo(double num1, double num2){
    return log(abs(num1-num2));
}

int main(){
    std::cout<<foo(10,10.5);
}
```

In this example, when you invoke `foo(10,10.5)`, you might expect the output to be `log(0.5)` or `-0.69`. Because the parameters of `abs` are not enclosed in parentheses, the output becomes `log(-20.5)` or `NaN`, which is unexpected and might lead to bugs. Polyspace flags the function-like macro definition.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M16-0-7

Undefined macro identifiers shall not be used in #if or #elif pre-processor directives, except as operands to the defined operator

## Description

### Rule Definition

*Undefined macro identifiers shall not be used in #if or #elif pre-processor directives, except as operands to the defined operator.*

### Rationale

If you attempt to use a macro identifier in a preprocessing directive, and you have not defined that identifier, then the preprocessor assumes that it has a value of zero. This value might not meet developer expectations.

### Polyspace Implementation

Polyspace flags an `#if` or `#elif` statement if it uses an undefined macro identifier.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Examples

### Macro Identifiers

```
#if M == 0           //Noncompliant
#endif

#if defined (M)      //Complaint
#if M == 0           //Executes only when M is defined
#endif
#endif

#if defined (M) && (M == 0)  //Compliant
//...
#endif
```

This example shows various uses of `M` in preprocessing directives:

- The first `#if` clause uses the undefined identifier `M`. Because `M` is undefined when this preprocessor directive is evaluated, the compiler assumes that `M` is zero, which results in unexpected results. Such a use of undefined identifiers is not compliant with this rule. Polyspace flags the `#if` statement.
- The second and third `#if` statements use the undefined identifier `M` as the operand to the `defined` operator. These use of undefined identifiers are compliant with this rule.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M16-0-8

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token

## Description

### Rule Definition

*If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.*

### Rationale

The # character precedes a preprocessor directive when it is the first character on a line. If the # character is not immediately followed by a preprocessor directive, the preprocessor directive might be malformed.

Preprocessor directives might be used to exclude portions of code from compilation. The compiler excludes code until it encounters an #else, #elif, or #endif preprocessor directive. If one of those preprocessor directives is malformed, the compiler continues excluding code beyond the intended end point, resulting in bugs and unexpected behavior which can be difficult to diagnose.

Avoid malformed preprocessor directives by placing the preprocessor token directly after a # token. Specifically, do not place any characters other than white space between the # token and preprocessor token in #else and #endif directives.

### Polyspace Implementation

Polyspace raises this defect when the # character is followed by any character that is not part of a properly formed preprocessor token. A preprocessor token that is preceded or followed by any character other than white space causes Polyspace to raise this defect. Polyspace raises this defect when a preprocessor token is badly formed due to misspelling or improper capitalization.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Poorly Formed Preprocessor Tokens Following # Character

```
#define TESTING_H       //Compliant

namespace Example
{
#ifndef TESTING_H       //Compliant
    // code here
#elseX;                 //Noncompliant
    // code here
#else;                  //Compliant
```

```
    // code here
#endnif                 //Noncompliant
    // code here

};
```

Because `elseX` is not a preprocessor directive and follows directly after the # character, Polyspace flags it as noncompliant.

`#endnif` is not a properly formed preprocessor directive. Polyspace flags it as noncompliant.

`#define TESTING_H`, `#ifndef TESTING_H`, and `#else` are properly formed preprocessor conditionals and are compliant with this rule.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M16-1-1

The defined pre-processor operator shall only be used in one of the two standard forms

## Description

### Rule Definition

*The defined pre-processor operator shall only be used in one of the two standard forms.*

### Rationale

The `defined` preprocessor operator checks whether an identifier is defined as a macro. In C, the only two permissible forms for this operator are:

- `defined (identifier)`
- `defined identifier`

Using any other form results in invalid code that compiler might not report. For instance, if you use expressions as arguments for the `defined` operator, the code is invalid. If the compiler does not report the invalid usage of `defined`, diagnosing the invalid code is difficult.

If your `#if` or similar preprocessor directives expand to create a `defined` statement, the code behavior is undefined. For instance:

```
#define DEFINED defined
#if DEFINED(X)
```

The `#if` preprocessor directive expands to form a `defined` operation. Depending on your environment, the code might behave in unexpected ways, leading to bugs that are difficult to diagnose.

To avoid invalid code, bugs, and undefined behavior, use only the permitted forms when using the `defined` operator.

### Polyspace Implementation

Polyspace flags incorrect usages of the `defined` operator, such as:

- The operator `defined` is used without an identifier.
- The operator `defined` appears after macro expansion.
- The operator `defined` is used with a complex expression.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use `defined` Operators With Identifiers

```
#if defined (X<Y)//Noncompliant
//...
#endif
#if defined (X) && defined (Y) &&(X<Y)//Compliant
//...
#endif
```

In this example, a block of code is conditionally executed only if the identifiers X and Y are defined and if X is smaller than Y. Constructing this condition by using an expression as the argument for the `defined` operator is not permissible and results in invalid code. Polyspace flags the impermissible `defined` statement. The permissible way to define such a condition is to use individual identifiers with `defined`.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M16-1-2

All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related

## Description

### Rule Definition

*All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related.*

### Rationale

You use preprocessor directives, such as `#if...#elif...#else...#endif`, to conditionally include or exclude blocks of code. If the different branches of such a directive reside in different source files, the code can be confusing. If all the branches are not included in a project, the code might behave in unexpected ways. To avoid confusion and unexpected behavior, keep the branches of a conditional preprocessor directive within the same source file.

### Polyspace Implementation

Polyspace raises a violation of this rule if either of these conditions are true:

- A corresponding `#if` directive cannot be found within a source file for every `#else`, `#elif`, or `#endif` directive.
- A corresponding `#endif` directive cannot be found within a source file for every `#if` directive.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Incomplete Conditional Preprocessor Directives

```
>//file1.h
#if !defined (FILE)
//.....
#elif //Noncompliant
//...///
```

```
//file2.h
#else //Noncompliant
//...
#endif //Noncompliant
///
```

In this example, a conditional directive is split across two source files.

- In `file1.h`, the `#if` directive has no corresponding `#endif` directive. Polyspace flags the block.

- In `file2.h`, the `#else` and `#endif` directives have no corresponding `#if` directive. Polyspace flags both directives.

### Check Information

**Group:** Preprocessing Directives
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

### See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

#### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M16-2-3

Include guards shall be provided

## Description

### Rule Definition

*Include guards shall be provided.*

### Rationale

When a translation unit contains a complex hierarchy of nested header files, it is possible for a particular header file to be included more than once, leading to confusion. If this multiple inclusion produces multiple or conflicting definitions, then your program can have undefined or erroneous behavior.

For instance, suppose that a header file contains:

```
#ifdef _WIN64
   int env_var;
#elseif
   long int env_var;
#endif
```

If the header file is contained in two inclusion paths, one that defines the macro `_WIN64` and another that undefines it, you can have conflicting definitions of `env_var`.

To avoid multiple inclusion of the same file, add include guards to the beginning of header files. Use either of these formats:

-
  ```
  <start-of-file>
  // Comments allowed here
  #if !defined ( identifier )
  #define identifier
  // Contents of file
  #endif
  <end-of-file>
  ```

-
  ```
  <start-of-file>
  // Comments allowed here
  #ifndef identifier
  #define identifier
  // Contents of file
  #endif
  <end-of-file>
  ```

### Polyspace Implementation

The checker raises a violation if a header file does not contain an include guard.

For instance, this code uses an include guard for the `#define` and `#include` statements. This code does not violate the rule:

```
// Contents of a header file
#ifndef FILE_H
```

```
#define FILE_H
#include "libFile.h"
#endif
```

If you use include guards that do not adhere to the suggested format, Polyspace flags them. For instance:

- You might mistakenly use different identifiers in the `#ifndef` and `#define` statements:

  ```
  #ifndef MACRO
  #define MICRO
  //...
  #endif
  ```
- You might inadvertently use `#ifdef` instead of `#ifndef` or omit the `#define` statement.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Missing or Incorrectly Formatted Include Guard

| file1.h | file2.h | mainfile.cpp |
|---|---|---|
| `#ifndef MACRO`<br>`#define MICRO`<br>`//...`<br>`#endif` | `#ifdef DO_INCLUDE`<br>`#define DO_INCLUDE`<br>`void foo();`<br>`#endif` | `#include"file1.h"`<br>`#include"file2.h"`<br>`int main(){`<br>`    return 0;`<br>`}` |

In this example, two header files are included in the file `mainfile.cpp`.

- The include guard in `file1.h` queries the definition of `MACRO` but conditionally defines a different identifier `MICRO`, perhaps inadvertently. This include guard is incorrectly formatted. Polyspace flags the file.
- The include guard in `file2.h` uses `#ifdef` instead of `#ifndef`. This include guard is incorrect and Polyspace flags the file.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M16-3-1

There shall be at most one occurrence of the # or ## operators in a single macro definition

## Description

### Rule Definition

*There shall be at most one occurrence of the # or ## operators in a single macro definition.*

### Rationale

The evaluation of the # and ## preprocessor operators does not have a specified execution order. When more than one occurrence of the # or ## operators exists in a single macro definition, it is unclear which preprocessor operator is executed first by the compiler. The uncertainty of execution order might result in developer confusion or unexpected macro calculations. Use only one of the # or ## preprocessor operators for each macro definition.

### Polyspace Implementation

Polyspace raises this defect whenever more than one instance of the # or ## operators is used in a single macro definition.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Multiple # and ## Operators Used in a Single Macro Definition

```
#define STRING(X) { #X }                          //Compliant
#define CONCAT(X, Y) {X ## Y}                     //Compliant
#define STRING_CONCAT(x, y) {#x ## y}             //Noncompliant
#define MULTI_CONCAT(x, y, xy, z) {x ## y ## z}   //Noncompliant
```

Because the macro `STRING_CONCAT` uses both the # and ## operators, Polyspace flags the macro as noncompliant.

Polyspace flags the macro `MULTI_CONCAT` as noncompliant because it uses multiple ## operators.

## Check Information

**Group:** Preprocessing Directives
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

**Topics**

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M16-3-2

The # and ## operators should not be used

## Description

### Rule Definition

*The # and ## operators should not be used.*

### Rationale

The evaluation of the # and ## preprocessor operators does not have a specified execution order. Different compilers might evaluate these operators in different order of execution. The uncertainty of execution order might result in developer confusion or unexpected macro calculations. When possible, avoid using the # and ## preprocessor operators.

### Polyspace Implementation

Polyspace raises this advisory when the # or ## operators are used in a macro definition.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Avoid Using # and ## Operators in Macro Definition

```
#define STRING(X) { #X }                             //Noncompliant
#define CONCAT(X, Y) {X ## Y}                        //Noncompliant
```

Because the macro `STRING(X)` uses the # operator, Polyspace flags the macro as noncompliant.

Polyspace flags the macro `CONCAT(X, Y)` as noncompliant because it uses the ## operator.

## Check Information

**Group:** Preprocessing Directives
**Category:** Advisory, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M17-0-2

The names of standard library macros and objects shall not be reused

## Description

### Rule Definition

*The names of standard library macros and objects shall not be reused.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Library Introduction
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M17-0-3

The names of standard library functions shall not be overridden

## Description

### Rule Definition

*The names of standard library functions shall not be overridden.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Library Introduction
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M17-0-5

The setjmp macro and the longjmp function shall not be used

## Description

### Rule Definition

*The setjmp macro and the longjmp function shall not be used.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Library Introduction
**Category:** Required, Automated

## Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M18-0-3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used

## Description

### Rule Definition

*The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Support Library
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M18-0-4

The time handling functions of library <ctime> shall not be used

## Description

### Rule Definition

*The time handling functions of library <ctime> shall not be used.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Support Library
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M18-0-5

The unbounded functions of library <cstring> shall not be used

## Description

### Rule Definition

*The unbounded functions of library <cstring> shall not be used.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see "Diagnose Why Coding Standard Violations Do Not Appear as Expected".

## Check Information

**Group:** Language Support Library
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

# AUTOSAR C++14 Rule M18-2-1

The macro offsetof shall not be used

## Description

### Rule Definition

*The macro offsetof shall not be used.*

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Support Library
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M18-7-1

The signal handling facilities of <csignal> shall not be used

## Description

### Rule Definition

*The signal handling facilities of <csignal> shall not be used.*

### Rationale

Signal handling functions such as `signal` contains undefined and implementation-specific behavior.

You have to be very careful when using `signal` to avoid these behaviors.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Check Information

**Group:** Language Support Library
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M19-3-1

The error indicator errno shall not be used

## Description

### Rule Definition

*The error indicator errno shall not be used.*

### Rationale

Observing this rule encourages the good practice of not relying on errno to check error conditions.

Checking errno is not sufficient to guarantee absence of errors. Functions such as fopen might not set errno on error conditions. Often, you have to check the return value of such functions for error conditions.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of errno

```
#include <cstdlib>
#include <cerrno>

void func (const char* str) {
    errno = 0;  // Noncompliant
    int i = atoi(str);
    if(errno != 0) { // Noncompliant
        //Handle Error
    }
}
```

The use of errno violates this rule. The function atoi is not required to set errno if the input string cannot be converted to an integer. Checking errno later does not safeguard against possible failures in conversion.

## Check Information

**Group:** Diagnostics Library
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

“Check for and Review Coding Standard Violations”

# AUTOSAR C++14 Rule M27-0-1

The stream input/output library <cstdio> shall not be used

## Description

### Rule Definition

*The stream input/output library <cstdio> shall not be used.*

### Rationale

Functions in `cstdio` such as `gets`, `fgetpos`, `fopen`, `ftell`, etc. have unspecified, undefined and implementation-defined behavior.

For instance:

- The `gets` function:

  ```
  char * gets ( char * buf );
  ```

  does not check if the number of characters provided at the standard input exceeds the buffer `buf`. The function can have unexpected behavior when the input exceeds the buffer.
- The `fopen` function has implementation-specific behavior related to whether it sets `errno` on errors or whether it accepts additional characters following the standard mode specifiers.

### Polyspace Implementation

Polyspace reports a violation of this rule if you use the functions from the `cstdio` library. Polyspace detects the use of these `cstdio` functions:

- File operation functions such as `remove` and `rename`.
- File access functions such as `fclose`,`fflush`, and `fopen`.
- Formatted input/output functions such as `fprintf`, `fscanf`, `printf`, and `scanf`.
- Character input output functions such as `fgetc`, `fgets`, `fputc`, and `getc`.
- Direct input/output functions such as `fread` and `fwrite`.
- File positioning functions such as `fgetpos` and `fsetpos`.
- Error handling functions such as `clearerr`, `ferror`, and `perror`.

### Troubleshooting

If you expect a rule violation but Polyspace does not report it, see “Diagnose Why Coding Standard Violations Do Not Appear as Expected”.

## Examples

### Use of `gets`

```
#include <cstdio>
```

```
void func()
{
    char array[10];
    fgets(array, sizeof array, stdin); //Noncompliant
}
```

The use of `fgets` violates this rule.

## Check Information

**Group:** Input Output Library
**Category:** Required, Automated

# Version History

**Introduced in R2019a**

## See Also

`Check AUTOSAR C++ 14 (-autosar-cpp14)`

### Topics

"Check for and Review Coding Standard Violations"

27

# Common Weakness Enumeration (CWE)

# CWE Rule 14

Compiler Removal of Code to Clear Buffers

## Description

### Rule Description

*Sensitive memory is cleared according to the source code, but compiler optimizations leave the memory untouched when it is not read from again, aka "dead store removal."*

### Polyspace Implementation

The rule checker checks for **Uncertain memory cleaning**.

## Examples

### Uncertain memory cleaning

#### Issue

This issue occurs when you clean memory by writing `0` into it but the memory is not read after the cleaning operation. Because the memory is not read after the cleaning, a compiler might skip the cleaning operation when optimizing the code. For instance, consider this code:

```
void foo(){
    void* buffer = getSensitiveData();
    //...
    memset(buffer, 0, sizeof(buffer));
}
```

Here, `buffer` might contain sensitive data. The call to `memset()` is intended to scrub the data. Because buffer is not read after the scrubbing, the compiler optimization process might ignore the `memset()` command. The data in `buffer` might remain uncleaned. Polyspace flags such uncertain memory cleaning. Polyspace does not raise this defect if either of these conditions are true:

- The return value of the memory cleaning function is assigned to a variable.
- The memory cleaning function scrubs a global object.

#### Risk

When a variable is written with no further read, compilers might consider it as a deadstore. To optimize the code, the compiler might remove any assignments to deadstore variables.

If such code optimization removes code that scrubs sensitive information from memory, then sensitive information might remain uncleaned in your system. An attacker can then access the sensitive information and use it to further erode the protection mechanisms in your code.

#### Fix

When scrubbing memory, use code that the compiler cannot remove. For instance, instead of `memset`, use secure functions, such as `memset_s` or `fill_n`. If you use an older version of C or C++ where these functions are unavailable, consider updating your code to a more recent version. Alternatively, you might implement your own secure memory-scrubbing function. When using your own implementation, review the compiled code to make sure that the memory-cleaning operations are untouched by the compiler.

#### Example — Avoid Uncertainty in Cleaning Sensitive Memory

```
#include <string.h>

extern int getSensitiveData(char*, size_t);
extern int writeDB(char*, char*);
char Gbuffer[64];
void Job(char *key) {
    char buffer[64];
    if (getSensitiveData(buffer, sizeof(buffer))) {
        if (writeDB(key, buffer)) {
            // Record data
        }
    }
    (void) memset(buffer, 0, sizeof(buffer)); // Defect
}

void manageGBuffer(char *key) {
    //...
    memset(Gbuffer, 0, sizeof(Gbuffer)); // No Defect
}
```

In this example, the command `memset()` is used in `Job()` to clean the memory in `buffer`. Because `buffer` is not read after the cleaning, the compiler might remove this code when optimizing the code. Polyspace flags the uncertain cleaning. In the function `manage()`, `memset()` is used to clean the global array `Gbuffer`. Polyspace does not flag this cleaning operation.

#### Correction — Use Secure Functions to Clean Sensitive Memory

To fix this defect, use secure functions, such as `memset_s()`, to clean uncertain memory.

```
#include <string.h>

extern int getSensitiveData(char*, size_t);
extern int writeDB(char*, char*);

void Job(char *key) {
    char buffer[64];
    if (getSensitiveData(buffer, sizeof(buffer))) {
        if (writeDB(key, buffer)) {
            // Record data
        }
    }
    memset_s(buffer, 0, sizeof(buffer)); // No Defect
}
```

## Check Information

**Category:** Others

## Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-14

# CWE Rule 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

## Description

### Rule Description

*The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access out of bounds**
- **Array access with tainted index**
- **Buffer overflow from incorrect string format specifier**
- **Cast to pointer pointing to object of different type**
- **Destination buffer overflow in string manipulation**
- **Destination buffer underflow in string manipulation**
- **Invalid use of standard library memory routine**
- **Invalid use of standard library routine**
- **Invalid use of standard library string routine**
- **Mismatch between data length and size**
- **Missing null in string array**
- **Pointer access out of bounds**
- **Pointer dereference with tainted offset**
- **Possible misuse of sizeof**
- **Reading memory reallocated from object of another type without reinitializing first**
- **Tainted NULL or non-null-terminated string**
- **Use of dangerous standard function**
- **Use of indeterminate string**

## Examples

### Array access out of bounds

#### Issue

This issue occurs when an array index falls outside the range `[0...array_size-1]` during array access.

#### Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }
    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

**Correction — Keep Array Index Within Array Bounds**

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

### Array access with tainted index

**Issue**

This issue occurs when you access an array by using an index that is obtained from unsecure sources and which has not been validated.

**Risk**

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

**Fix**

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Use Index to Return Buffer Value

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];//Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The index `num` is obtained from an unsecure source and the function `taintedarrayindex` does not check to see if `num` is inside the range of `tab`.

#### Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

### Buffer overflow from incorrect string format specifier

#### Issue

This issue occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

#### Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

#### Fix

Use a format specifier that is compatible with the memory buffer size.

**Example — Memory Buffer Overflow**

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

**Correction — Use Smaller Precision in Format Specifier**

One possible correction is to read a smaller number of elements into the buffer.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

### Cast to pointer pointing to object of different type

**Issue**

The issue occurs when you perform a cast between a pointer to an object type and a pointer to a different object type.

**Risk**

If a pointer to an object is cast into a pointer to a different object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.

Even if the conversion produces a pointer that is correctly aligned, the behavior can be undefined if the pointer is used to access an object.

Exception: You can convert a pointer to object type into a pointer to one of these types:

- `char`
- `signed char`
- `unsigned char`

**Example - Cast to Pointer Pointing to Object of Wider Type**

```
signed   char *p1;
unsigned int *p2;

void foo(void){
  p2 = ( unsigned int * ) p1;     /* Non-compliant */
}
```

In this example, `p1` can point to a `signed char` object. However, `p1` is cast to a pointer that points to an object of wider type, `unsigned int`.

**Example - Cast to Pointer Pointing to Object of Narrower Type**

```
extern unsigned int read_value ( void );
extern void display ( unsigned int n );
```

```
void foo ( void ){
  unsigned int u = read_value ( );
  unsigned short *hi_p = ( unsigned short * ) &u;    /* Non-compliant  */
  *hi_p = 0;
  display ( u );
}
```

In this example, u is an unsigned int variable. &u is cast to a pointer that points to an object of narrower type, unsigned short.

On a big-endian machine, the statement *hi_p = 0 attempts to clear the high bits of the memory location that &u points to. But, from the result of display(u), you might find that the high bits have not been cleared.

**Example - Cast Adding a Type Qualifier**

```
const short *p;
const volatile short *q;
void foo (void){
  q = ( const volatile short * ) p;  /* Compliant */
}
```

In this example, both p and q can point to short objects. The cast between them adds a volatile qualifier only and is therefore compliant.

### Destination buffer overflow in string manipulation

**Issue**

This issue occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function sprintf(char* buffer, const char* format), you use a constant string format of greater size than buffer.

**Risk**

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

**Fix**

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use sprintf to write formatted data to a string, use snprintf, _snprintf or sprintf_s instead to enforce length control. Alternatively, use asprintf to automatically allocate the memory required for the destination buffer.
- If you use vsprintf to write formatted data from a variable argument list to a string, use vsnprintf or vsprintf_s instead to enforce length control.
- If you use wcscpy to copy a wide string, use wcsncpy, wcslcpy, or wcscpy_s instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example — Buffer Overflow in sprintf Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

#### Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Destination buffer underflow in string manipulation

#### Issue

This issue occurs when certain string manipulation functions write to their destination buffer argument at a negative offset from the beginning of the buffer.

For instance, for the function `sprintf(char* buffer, const char* format)`, you obtain the `buffer` from an operation `buffer = (char*)arr; ... buffer += offset;`. `arr` is an array and `offset` is a negative value.

#### Risk

Buffer underflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer underflow also introduces the risk of code injection.

#### Fix

If the destination buffer argument results from pointer arithmetic, see if you are decrementing a pointer. Fix the pointer decrement by modifying either the original value before decrement or the decrement value.

#### Example — Buffer Underflow in sprintf Use

```
#include <stdio.h>
#define offset -2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

In this example, &buffer[offset] is at a negative offset from the memory allocated to buffer.

#### Correction — Change Pointer Decrementer

One possible correction is to change the value of offset.

```
#include <stdio.h>
#define offset 2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

### Invalid use of standard library memory routine

#### Issue

This issue occurs when a memory library function is called with invalid arguments. For instance, the memcpy function copies to an array that cannot accommodate the number of bytes copied.

#### Risk

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example — Invalid Use of Standard Library Memory Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);
  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

**Correction — Call Function with Valid Arguments**

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

### Invalid use of standard library routine

**Issue**

This issue occurs when you use invalid arguments with a function from the standard library. This defect picks up errors related to other functions not covered by float, integer, memory, or string standard library routines.

**Risk**

Invalid arguments to a standard library function result in undefined behavior.

**Fix**

The fix depends on the root cause of the defect. For instance, the argument to a `printf` function can be `NULL` because a pointer was initialized with `NULL` and the initialization value was not overwritten along a specific execution path.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Calling printf Without a String

```
#include <stdio.h>
#include <stdlib.h>

void print_null(void) {

  printf(NULL);
}
```

The function `printf` takes only string input arguments or format specifiers. In this function, the input value is NULL, which is not a valid string.

#### Correction — Use Compatible Input Arguments

One possible correction is to change the input arguments to fit the requirements of the standard library routine. In this example, the input argument was changed to a character.

```
#include <stdio.h>

void print_null(void) {
    char zero_val = '0';
    printf((const char*)zero_val);
}
```

### Invalid use of standard library string routine

#### Issue

This issue occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example — Invalid Use of Standard Library String Routine Error**

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

**Correction — Use Valid Arguments**

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Mismatch between data length and size

**Issue**

This issue occurs when you do not check the length argument and data buffer argument of memory copying functions such as `memcpy`, `memset`, or `memmove`, to protect against buffer overflows.

**Risk**

If an attacker can manipulate the data buffer or length argument, the attacker can cause buffer overflow by making the actual data size smaller than the length.

This mismatch in length allows the attacker to copy memory past the data buffer to a new location. If the extra memory contains sensitive information, the attacker can now access that data.

This defect is similar to the SSL Heartbleed bug.

**Fix**

When copying or manipulating memory, compute the length argument directly from the data so that the sizes match.

**Example — Copy Buffer of Data**

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    memcpy(&(beta.data[num]), os->data + 2, length);

    return(1);
}
```

This function copies the buffer `alpha` into a buffer `beta`. However, the `length` variable is not related to `data+2`.

**Correction — Check Buffer Length**

One possible correction is to check the length of your buffer against the maximum value minus 2. This check ensures that you have enough space to copy the data to the `beta` structure.

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    if (length<(os->max -2)) {
        memcpy(&(beta.data[num]), os->data + 2, length);
    }

    return(1);
}
```

### Missing null in string array

#### Issue

This issue occurs when a string does not have enough space to terminate with a null character `'\0'`.

This defect applies only for projects in C.

#### Risk

A buffer overflow can occur if you copy a string to an array without assuming the implicit null terminator.

#### Fix

If you initialize a character array with a literal, avoid specifying the array bounds.

```
char three[]  = "THREE";
```

The compiler automatically allocates space for a null terminator. In the preceding example, the compiler allocates sufficient space for five characters and a null terminator.

If the issue occurs after initialization, you might have to increase the size of the array by one to account for the null terminator.

In certain circumstances, you might want to initialize the character array with a sequence of characters instead of a string. In this situation, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example — Array size is too small

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE";
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

#### Correction — Increase Array Size

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

#### Correction — Change Initialization Method

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

### Pointer access out of bounds

#### Issue

This issue occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

#### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.

- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example — Pointer access out of bounds error**

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

**Correction — Check Pointer Stays Within Bounds**

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

### Pointer dereference with tainted offset

**Issue**

This issue occurs when a pointer dereference uses an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

**Risk**

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

#### Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Dereference Pointer Array

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset];//Noncompliant
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of `offset`. Continue with the pointer dereferencing only if `offset` is inside the valid range.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

### Possible misuse of sizeof

#### Issue

This issue occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:

  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

#### Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

#### Fix

Possible fixes are:

- Do not use the sizeof operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the sizeof operator carefully to determine the number argument of functions such as strncmp or wcsncpy. For instance, for wide string functions such as wcsncpy, use the number of wide characters as argument instead of the number of bytes.

#### Example — sizeof Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
    }
}
```

In this example, sizeof(a) returns the size of the pointer a and not the array size.

#### Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

### Reading memory reallocated from object of another type without reinitializing first

#### Issue

This issue occurs when you do the following in sequence:

1. Reallocate memory to an object with a type that is different from the original allocation.

   For instance, in this code snippet, a memory originally allocated to a pointer with type struct A* is reallocated to a pointer with type struct B*:

   ```
   struct A;
   struct B;

   struct A *Aptr = (struct A*) malloc(sizeof(struct A));
   struct B *Bptr = (struct B*) realloc(Aptr, sizeof(struct B));
   ```

**2** Read from this reallocated memory without reinitializing the memory first.

Read accesses on the pointer to the reallocated memory can happen through pointer dereference or array indexing. Passing the pointer to a function that takes a pointer to a `const`-qualified object as the corresponding parameter also counts as a read access.

#### Risk

Reading from reallocated memory that has not been reinitialized leads to undefined behavior.

#### Fix

Reinitialize memory after reallocation and before the first read access.

The checker considers any write access on the pointer to the reallocated memory as satisfying the reinitialization requirement (even if the object might only be partially reinitialized). Write accesses on the pointer to the reallocated memory can happen through pointer dereference or array indexing. Passing the pointer to a function that takes a pointer to a non-`const`-qualified object as the corresponding parameter also counts as a write access.

#### Example – Noncompliant: Reading from Reallocated Memory Without Reinitializing First

```
#include<stdlib.h>

struct group {
    char *groupFirst;
    int groupSize;
};

struct groupWithID {
    int groupID;
    char *groupFirst;
    int groupSize;
};

char* readName();
int readSize();

void createGroup(int nextAvailableID) {
    struct group *aGroup;
    struct groupWithID *aGroupWithID;

    aGroup = (struct group*) malloc(sizeof(struct group));

    if(!aGroup) {
        /*Handle error*/
    }

    aGroup->groupFirst = readName();
    aGroup->groupSize  = readSize();

    /* Reassign to group with ID */
    aGroupWithID = (struct groupWithID*) realloc(aGroup, sizeof(struct groupWithID));
    if(!aGroupWithID) {
        free(aGroup);
        /*Handle error*/
    }
    if(aGroupWithID -> groupSize > 0) { /* Noncompliant */
        /* ... */
    }

    /* ...*/
    free(aGroupWithID);
}
```

In this example, the memory allocated to a `group*` pointer using the `malloc` function is reallocated to a `groupWithID*` pointer using the `realloc` function. There is a read access on the reallocated memory before the memory is reinitialized.

**Correction – Reinitialize Memory After Reallocation and Before First Read**

Reinitialize the memory assigned to the `groupWithID*` pointer before the first read access. All bits of the memory can be reinitialized using the `memset` function.

```
#include<stdlib.h>
#include<string.h>

struct group {
    char *groupFirst;
    int groupSize;
};

struct groupWithID {
    int groupID;
    char *groupFirst;
    int groupSize;
};

char* readName();
int readSize();

void createGroup(int nextAvailableID) {
    struct group *aGroup;
    struct groupWithID *aGroupWithID;

    aGroup = (struct group*) malloc(sizeof(struct group));

    if(!aGroup) {
        /*Handle error*/
    }

    aGroup->groupFirst = readName();
    aGroup->groupSize  = readSize();

    /* Reassign to group with ID */
    aGroupWithID = (struct groupWithID*) realloc(aGroup, sizeof(struct groupWithID));
    if(!aGroupWithID) {
        free(aGroup);
        /*Handle error*/
    }

    memset(aGroupWithID, 0 , sizeof(struct groupWithID));
    /* Reinitialize group */
    if(aGroupWithID -> groupSize > 0) {
        /* ... */
    }

    /* ...*/
    free(aGroupWithID);
}
```

### Tainted NULL or non-null-terminated string

#### Issue

This issue occurs when strings from unsecure sources are used in string manipulation routines that implicitly dereference the string buffer, for instance, `strcpy` or `sprintf`.

**Tainted NULL or non-null-terminated string** raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

#### Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

#### Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Getting String from Input

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));//Noncompliant
    print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

**Correction — Validate the Data**

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // Noncompliant-TAINTED_STRING only flagged here
        // - string is not null
        // - string has a positive and limited size
        // - TAINTED_STRING on strlen used as a firewall
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

**Correction — Validate the Data**

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

```
void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

### Use of dangerous standard function

#### Issue

This issue occurs when your code uses standard functions that write data to a buffer in a way that can result in buffer overflows.

The following table lists dangerous standard functions, the risks of using each function, and what function to use instead. The checker flags:

- Any use of an inherently dangerous function.
- An use of a possibly dangerous function only if the size of the buffer to which data is written can be determined at compile time. The checker does not flag an use of such a function with a dynamically allocated buffer.

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `gets` | Inherently dangerous — You cannot control the length of input from the console. | `fgets` |
| `std::cin::operator>>` and `std::wcin::operator>>` | Inherently dangerous — You cannot control the length of input from the console. | Preface calls to `cin` by `cin.width` to control the input length. This method can result in truncated input.<br>To avoid potential buffer overflow and truncated input, use `std::string` objects as destinations for >> operator. |

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `strcpy` | Possibly dangerous — If the size of the destination buffer is too small to accommodate the source buffer and a null terminator, a buffer overflow might occur. | Use the function `strlen()` to determine the size of the source buffer, and allocate sufficient memory so that the destination buffer can accommodate the source buffer and a null terminator. Instead of `strcpy`, use the function `strncpy`. |
| `stpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `stpncpy` |
| `lstrcpy` or `StrCpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy` |
| `strcat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `strncat`, `strlcat`, or `strcat_s` |
| `lstrcat` or `StrCat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat` |
| `wcpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcpncpy` |
| `wcscat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `wcsncat`, `wcslcat`, or `wcncat_s` |
| `wcscpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcsncpy` |
| `sprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `snprintf` |
| `vsprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `vsnprintf` |

**Risk**

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

**Fix**

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Example — Using sprintf

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

### Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

### Use of indeterminate string

#### Issue

This issue occurs when you do not check if a write operation using an fgets-family function such as:

```
char * fgets(char* buf, int n, FILE *stream)
```

succeeded and the buffer written has valid content, or you do not reset the buffer on failure. You then perform an operation that assumes a buffer with valid content. For instance, if the buffer with possibly indeterminate content is buf (as shown above), the checker raises a defect if:

- You pass buf as argument to standard functions that print or manipulate strings or wide strings.
- You return buf from a function.
- You pass buf as argument to external functions with parameter type const char * or const wchar_t *.
- You read buf as buf[index] or *(buf + offset), where index or offset is a numerical value representing the distance from the beginning of the buffer.

#### Risk

If an fgets-family function fails, the content of its output buffer is indeterminate. Use of such a buffer has undefined behavior and can result in a program that stops working or other security vulnerabilities.

#### Fix

Reset the output buffer of an fgets-family function to a known string value when the function fails.

#### Example — Output of fgets() Passed to External Function

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func(void) {
    char buf[SIZE20];

    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* 'buf' may contain an indeterminate string.  */
        ;
    }
    /* 'buf passed to external function */
    display_text(buf);
}
```

In this example, the output buf is passed to the external function display_text(), but its value is not reset if fgets() fails.

#### Correction — Reset `fgets()` Output on Failure

If `fgets()` fails, reset `buf` to a known value before you pass it to an external function.

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

#define SIZE20 20

extern void display_text(const char *txt);

void func1(void) {
    char buf[SIZE20];
    /* Check fgets() error */
    if (fgets (buf, sizeof (buf), stdin) == NULL)
    {
        /* value of 'buf' reset after fgets() failure. */
        buf[0] = '\0';
    }
    /* 'buf' passed to external function */
    display_text(buf);
}
```

### Check Information

**Category:** Others

## Version History

**Introduced in R2023a**

### See Also

`Check CWE (-cwe)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

CWE-119

# CWE Rule 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

## Description

### Rule Description

*The program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Destination buffer overflow in string manipulation**
- **Invalid use of standard library memory routine**
- **Invalid use of standard library string routine**
- **Tainted NULL or non-null-terminated string**

## Examples

### Destination buffer overflow in string manipulation

#### Issue

This issue occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

**Example — Buffer Overflow in sprintf Use**

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

**Correction — Use `snprintf` Instead of `sprintf`**

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Invalid use of standard library memory routine

**Issue**

This issue occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

**Risk**

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

**Fix**

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example — Invalid Use of Standard Library Memory Routine Error**

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

**Correction — Call Function with Valid Arguments**

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

### Invalid use of standard library string routine

**Issue**

This issue occurs when a string library function is called with invalid arguments.

**Risk**

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

**Fix**

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

#### Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Tainted NULL or non-null-terminated string

#### Issue

This issue occurs when strings from unsecure sources are used in string manipulation routines that implicitly dereference the string buffer, for instance, `strcpy` or `sprintf`.

**Tainted NULL or non-null-terminated string** raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

#### Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

#### Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Getting String from Input

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));//Noncompliant
    print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

### Correction — Validate the Data

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // Noncompliant-TAINTED_STRING only flagged here
        // - string is not null
        // - string has a positive and limited size
        // - TAINTED_STRING on strlen used as a firewall
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

### Correction — Validate the Data

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}
```

```
int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

## Check Information

**Category:** Memory Buffer Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-120

# CWE Rule 121

Stack-based Buffer Overflow

## Description

### Rule Description

*A stack-based buffer overflow condition is a condition where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access with tainted index**
- **Destination buffer overflow in string manipulation**
- **Invalid use of standard library memory routine**
- **Invalid use of standard library string routine**

## Examples

### Array access with tainted index

#### Issue

This issue occurs when you access an array by using an index that is obtained from unsecure sources and which has not been validated.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

#### Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Use Index to Return Buffer Value

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];//Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The index `num` is obtained from an unsecure source and the function `taintedarrayindex` does not check to see if `num` is inside the range of `tab`.

#### Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

### Destination buffer overflow in string manipulation

#### Issue

This issue occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

**Example — Buffer Overflow in sprintf Use**

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

**Correction — Use `snprintf` Instead of `sprintf`**

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Invalid use of standard library memory routine

**Issue**

This issue occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

**Risk**

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

**Fix**

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example — Invalid Use of Standard Library Memory Routine Error**

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

**Correction — Call Function with Valid Arguments**

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

### Invalid use of standard library string routine

**Issue**

This issue occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

#### Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Check Information

**Category:** Others

## Version History

**Introduced in R2023a**

### See Also

`Check CWE (-cwe)`

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

CWE-121

# CWE Rule 122

Heap-based Buffer Overflow

## Description

### Rule Description

*A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as malloc().*

### Polyspace Implementation

The rule checker checks for these issues:

- **Destination buffer overflow in string manipulation**
- **Pointer dereference with tainted offset**

## Examples

### Destination buffer overflow in string manipulation

#### Issue

This issue occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

**Example — Buffer Overflow in sprintf Use**

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

**Correction — Use `snprintf` Instead of `sprintf`**

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Pointer dereference with tainted offset

**Issue**

This issue occurs when a pointer dereference uses an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

**Risk**

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

**Fix**

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

**Extend Checker**

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of

Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Dereference Pointer Array

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset];//Noncompliant
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of `offset`. Continue with the pointer dereferencing only if `offset` is inside the valid range.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-122

# CWE Rule 123

Write-what-where Condition

## Description

### Rule Description

*Any condition where the attacker has the ability to write an arbitrary value to an arbitrary location, often as the result of a buffer overflow.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access with tainted index**
- **Invalid use of standard library memory routine**
- **Pointer dereference with tainted offset**
- **Tainted NULL or non-null-terminated string**

## Examples

### Array access with tainted index

#### Issue

This issue occurs when you access an array by using an index that is obtained from unsecure sources and which has not been validated.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

#### Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

**Example — Use Index to Return Buffer Value**

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];//Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The index `num` is obtained from an unsecure source and the function `taintedarrayindex` does not check to see if `num` is inside the range of `tab`.

**Correction — Check Range Before Use**

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

### Invalid use of standard library memory routine

**Issue**

This issue occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

**Risk**

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

**Fix**

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example — Invalid Use of Standard Library Memory Routine Error**

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

**Correction — Call Function with Valid Arguments**

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

### Pointer dereference with tainted offset

**Issue**

This issue occurs when a pointer dereference uses an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

#### Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See "Sources of Tainting in a Polyspace Analysis". To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Dereference Pointer Array

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset];//Noncompliant
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of `offset`. Continue with the pointer dereferencing only if `offset` is inside the valid range.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

### Tainted NULL or non-null-terminated string

#### Issue

This issue occurs when strings from unsecure sources are used in string manipulation routines that implicitly dereference the string buffer, for instance, `strcpy` or `sprintf`.

**Tainted NULL or non-null-terminated string** raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

#### Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

#### Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of

Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

**Example — Getting String from Input**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));//Noncompliant
    print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

**Correction — Validate the Data**

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // Noncompliant-TAINTED_STRING only flagged here
        // - string is not null
        // - string has a positive and limited size
        // - TAINTED_STRING on strlen used as a firewall
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

#### Correction — Validate the Data

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

## Check Information

**Category:** Memory Buffer Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

CWE-123

# CWE Rule 124

Buffer Underwrite ('Buffer Underflow')

## Description

### Rule Description

*The software writes to a buffer using an index or pointer that references a memory location prior to the beginning of the buffer.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access with tainted index**
- **Destination buffer underflow in string manipulation**
- **Pointer dereference with tainted offset**

## Examples

### Array access with tainted index

#### Issue

This issue occurs when you access an array by using an index that is obtained from unsecure sources and which has not been validated.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

#### Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Use Index to Return Buffer Value

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];//Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The index `num` is obtained from an unsecure source and the function `taintedarrayindex` does not check to see if `num` is inside the range of `tab`.

#### Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

### Destination buffer underflow in string manipulation

#### Issue

This issue occurs when certain string manipulation functions write to their destination buffer argument at a negative offset from the beginning of the buffer.

For instance, for the function `sprintf(char* buffer, const char* format)`, you obtain the `buffer` from an operation `buffer = (char*)arr; ... buffer += offset;`. `arr` is an array and `offset` is a negative value.

#### Risk

Buffer underflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer underflow also introduces the risk of code injection.

#### Fix

If the destination buffer argument results from pointer arithmetic, see if you are decrementing a pointer. Fix the pointer decrement by modifying either the original value before decrement or the decrement value.

**Example — Buffer Underflow in sprintf Use**

```
#include <stdio.h>
#define offset -2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

In this example, `&buffer[offset]` is at a negative offset from the memory allocated to `buffer`.

**Correction — Change Pointer Decrementer**

One possible correction is to change the value of `offset`.

```
#include <stdio.h>
#define offset 2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

### Pointer dereference with tainted offset

**Issue**

This issue occurs when a pointer dereference uses an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

**Risk**

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

**Fix**

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

**Extend Checker**

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of

Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Dereference Pointer Array

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset];//Noncompliant
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of `offset`. Continue with the pointer dereferencing only if `offset` is inside the valid range.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

## Check Information

**Category:** Memory Buffer Errors

# Version History

**Introduced in R2023a**

## See Also

Check CWE (-cwe)

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-124

# CWE Rule 125

Out-of-bounds Read

## Description

### Rule Description

*The software reads data past the end, or before the beginning, of the intended buffer.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access out of bounds**
- **Array access with tainted index**
- **Buffer overflow from incorrect string format specifier**
- **Destination buffer overflow in string manipulation**
- **Destination buffer underflow in string manipulation**
- **Invalid use of standard library memory routine**
- **Invalid use of standard library string routine**
- **Missing null in string array**
- **Pointer access out of bounds**
- **Pointer dereference with tainted offset**
- **Possible misuse of sizeof**
- **Tainted NULL or non-null-terminated string**

## Examples

### Array access out of bounds

#### Issue

This issue occurs when an array index falls outside the range `[0...array_size-1]` during array access.

#### Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface" or "Interpret Bug Finder Results in Polyspace Access Web Interface".

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Extend Checker**

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

**Example — Array Access Out of Bounds Error**

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

#### Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

### Array access with tainted index

#### Issue

This issue occurs when you access an array by using an index that is obtained from unsecure sources and which has not been validated.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

#### Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

**Example — Use Index to Return Buffer Value**

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];//Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The index `num` is obtained from an unsecure source and the function `taintedarrayindex` does not check to see if `num` is inside the range of `tab`.

**Correction — Check Range Before Use**

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

### Buffer overflow from incorrect string format specifier

**Issue**

This issue occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

**Risk**

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

**Fix**

Use a format specifier that is compatible with the memory buffer size.

**Example — Memory Buffer Overflow**

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

#### Correction — Use Smaller Precision in Format Specifier

One possible correction is to read a smaller number of elements into the buffer.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

### Destination buffer overflow in string manipulation

#### Issue

This issue occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example — Buffer Overflow in sprintf Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, buffer can contain 20 char elements but fmt_string has a greater size.

#### Correction — Use snprintf Instead of sprintf

One possible correction is to use the snprintf function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Destination buffer underflow in string manipulation

#### Issue

This issue occurs when certain string manipulation functions write to their destination buffer argument at a negative offset from the beginning of the buffer.

For instance, for the function sprintf(char* buffer, const char* format), you obtain the buffer from an operation buffer = (char*)arr; ... buffer += offset;. arr is an array and offset is a negative value.

#### Risk

Buffer underflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer underflow also introduces the risk of code injection.

#### Fix

If the destination buffer argument results from pointer arithmetic, see if you are decrementing a pointer. Fix the pointer decrement by modifying either the original value before decrement or the decrement value.

#### Example — Buffer Underflow in sprintf Use

```
#include <stdio.h>
#define offset -2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

In this example, &buffer[offset] is at a negative offset from the memory allocated to buffer.

#### Correction — Change Pointer Decrementer

One possible correction is to change the value of offset.

```
#include <stdio.h>
#define offset 2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

### Invalid use of standard library memory routine

**Issue**

This issue occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

**Risk**

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

**Fix**

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example — Invalid Use of Standard Library Memory Routine Error**

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%9s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

#### Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%9s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

### Invalid use of standard library string routine

#### Issue

This issue occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>
  char* Copy_String(void)
  {
   char *res;
   char gbuffer[5],text[20]="ABCDEFGHIJKL";

   res=strcpy(gbuffer,text);
   /* Error: Size of text is less than gbuffer */

   return(res);
  }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

**Correction — Use Valid Arguments**

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Missing null in string array

**Issue**

This issue occurs when a string does not have enough space to terminate with a null character `'\0'`.

This defect applies only for projects in C.

**Risk**

A buffer overflow can occur if you copy a string to an array without assuming the implicit null terminator.

**Fix**

If you initialize a character array with a literal, avoid specifying the array bounds.

```
char three[]  = "THREE";
```

The compiler automatically allocates space for a null terminator. In the preceding example, the compiler allocates sufficient space for five characters and a null terminator.

If the issue occurs after initialization, you might have to increase the size of the array by one to account for the null terminator.

In certain circumstances, you might want to initialize the character array with a sequence of characters instead of a string. In this situation, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Array size is too small

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE";
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

#### Correction — Increase Array Size

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

#### Correction — Change Initialization Method

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

### Pointer access out of bounds

#### Issue

This issue occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

**Risk**

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

**Fix**

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface" or "Interpret Bug Finder Results in Polyspace Access Web Interface".

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example — Pointer access out of bounds error**

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

**Correction — Check Pointer Stays Within Bounds**

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

### Pointer dereference with tainted offset

#### Issue

This issue occurs when a pointer dereference uses an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

#### Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Dereference Pointer Array

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset];//Noncompliant
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of `offset`. Continue with the pointer dereferencing only if `offset` is inside the valid range.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

### Possible misuse of sizeof

#### Issue

This issue occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

#### Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

#### Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

#### Example — sizeof Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

#### Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

### Tainted NULL or non-null-terminated string

#### Issue

This issue occurs when strings from unsecure sources are used in string manipulation routines that implicitly dereference the string buffer, for instance, `strcpy` or `sprintf`.

**Tainted NULL or non-null-terminated string** raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

#### Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

#### Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Getting String from Input

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));//Noncompliant
    print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

#### Correction — Validate the Data

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // Noncompliant-TAINTED_STRING only flagged here
        // - string is not null
        // - string has a positive and limited size
        // - TAINTED_STRING on strlen used as a firewall
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

#### Correction — Validate the Data

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

## Check Information

**Category:** Memory Buffer Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-125

# CWE Rule 126

Buffer Over-read

## Description

### Rule Description

*The software reads from a buffer using buffer access mechanisms such as indexes or pointers that reference memory locations after the targeted buffer.*

### Polyspace Implementation

The rule checker checks for **Buffer overflow from incorrect string format specifier**.

## Examples

### Buffer overflow from incorrect string format specifier

#### Issue

This issue occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

#### Risk

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

#### Fix

Use a format specifier that is compatible with the memory buffer size.

#### Example — Memory Buffer Overflow

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

#### Correction — Use Smaller Precision in Format Specifier

One possible correction is to read a smaller number of elements into the buffer.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-126

# CWE Rule 127

Buffer Under-read

## Description

### Rule Description

*The software reads from a buffer using buffer access mechanisms such as indexes or pointers that reference memory locations prior to the targeted buffer.*

### Polyspace Implementation

The rule checker checks for **Buffer overflow from incorrect string format specifier**.

## Examples

### Buffer overflow from incorrect string format specifier

**Issue**

This issue occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

**Risk**

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

**Fix**

Use a format specifier that is compatible with the memory buffer size.

**Example — Memory Buffer Overflow**

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

**Correction — Use Smaller Precision in Format Specifier**

One possible correction is to read a smaller number of elements into the buffer.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-127

# CWE Rule 128

Wrap-around Error

## Description

### Rule Description

*Wrap around errors occur whenever a value is incremented past the maximum value for its type and therefore "wraps around" to a very small, negative, or undefined value.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Integer constant overflow**
- **Integer conversion overflow**
- **Integer overflow**
- **Tainted sign change conversion**
- **Unsigned integer constant overflow**
- **Unsigned integer conversion overflow**
- **Unsigned integer overflow**

## Examples

### Integer constant overflow

#### Issue

This issue occurs in the following cases:

- You assign a compile-time integer constant to a signed integer variable whose data type cannot accommodate the value.

  See “Constant Overflows from Assignments” on page 7-17.
- You use an `enum` value that cannot be accommodated by the underlying type of the `enum` (and the underlying type is signed). For most C compilers, the default underlying type is `signed int` (based on the C standard).

  See “Constant Overflows from enum Values” on page 7-17.
- You perform a binary operation involving two integer constants that results in an overflow, that is, a value outside the range allowed by the data type that the operation uses. A binary operation with integer constants uses the `signed int` data type (unless you use modifiers such as `u` or `L`).

  See “Constant Overflows from Binary Operations” on page 7-18.

An `n`-bit signed integer holds values in the range [$-2^{n-1}$, $2^{n-1}-1$]. For instance, `c` is an 8-bit signed `char` variable that cannot hold the value 255.

```
signed char c = 255;
```

This defect checker depends on the following options:

- `Target processor type (-target)`: Determines the sizes of fundamental types.
- `Enum type definition (-enum-type-definition)`: Determines the underlying types of enumerations.
- `Compiler (-compiler)`: Impacts the interpretation of code.

You do not see the defect in these situations:

- Creation of new constants from `const` variables (for specific compilers only).

  Different compilers might define compile-time constants differently. In the following code, `c+1` is considered a compile-time constant by GCC compilers, but not by the standard C compiler:

  ```
  const int16_t c = 32767;
  int16_t y = c + 1;
  ```

  Whether you see a violation of this check on `y` might depend on your compiler.
- Bitwise `NOT` operation.

  Polyspace does not raise this violation when you perform a bitwise `NOT` operation.

**Risk**

The default behavior for constant overflows can vary between compilers and platforms. Retaining constant overflows can reduce the portability of your code.

Even if your compilers wraps around overflowing constants with a warning, the wrap-around behavior can be unintended and cause unexpected results.

**Fix**

Check if the constant value is what you intended. If the value is correct, use a different, possibly wider, data type for the variable.

**Example — Constant Overflows from Assignments**

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    char c1 = MAX_UNSIGNED_CHAR;
    char c2 = MAX_SIGNED_CHAR+1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow. To reproduce these defects, use a `Target processor type (-target)` where `char` is signed by default.

**Correction — Use Different Data Type**

One possible correction is to use a different data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
```

```
    unsigned char c1 = MAX_UNSIGNED_CHAR;
    unsigned char c2 = MAX_SIGNED_CHAR+1;
}
```

**Example — Constant Overflows from enum Values**

```
enum {
  a=0x7fffffff,
  b,
  c=0xffffffff
} MyEnumA;
```

In this example, the `enum` has an underlying type `int`. The `int` type can accommodate values in the range [-$2^{31}$, $2^{31}$-1]. However, the value of the enumerator `b` is 0x80000000 or $2^{31}$ (one more than the previous value `a`). This value falls outside the allowed range of `int`.

The value of `c`, that is 0xffffffff or $2^{32}$-1, is even larger and also causes an overflow.

To see this defect:

- Specify `Compiler (-compiler)` as `generic`.
- Specify `Source code language (-lang)` as `c`.

**Example — Constant Overflows from Binary Operations**

```
const unsigned int K_ATM_Label_Ram_init_value [] = {
     0x06 | ( 3 << 29) ,
     0x80 | ( 9 << 29) ,
     ( 2 << 31 )        ,
};
```

In this example, two of the shift operations result in values that cannot be accommodated by the `signed int` data type. The `signed int` data type can accommodate values in the range [-$2^{31}$, $2^{31}$-1]. The operation:

- `9 << 29` results in the value $2^{32}$+536870912.
- `2 << 31` results in the value $2^{32}$.

Note that even though the result is assigned to an `unsigned int` variable, the overflow detection uses the underlying type of the binary operation, that is, `signed int`.

### Integer conversion overflow

#### Issue

This issue occurs when converting an integer to a smaller integer type. If the variable does not have enough bytes to represent the original value, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer conversion overflows result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Converting from int to char

```
char convert(void) {

    int num = 1000000;

    return (char)num;
}
```

In the return statement, the integer variable `num` is converted to a `char`. However, an 8-bit or 16-bit character cannot represent 1000000 because it requires at least 20 bits. So the conversion operation overflows.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number.

```
long convert(void) {

    int num = 1000000;

    return (long)num;
}
```

### Integer overflow

#### Issue

This issue occurs when an operation on integer variables results in values that cannot be represented by the data type that the operation uses. This data type depends on the operand types and determines the number of bytes allocated for storing the result, thus constraining the range of allowed values.

Note that:

- The data type used to determine an overflow is based on the operand data types. If you then assign the result of an operation to another variable, a different checker, `Integer conversion overflow`, determines if the value assigned also overflows the variable assigned to. For instance, in an operation such as:

  ```
  res = x + y;
  ```

  This checker checks for an overflow based on the types of `x` and `y`, and not on the type of `res`. The checker for integer conversion overflows then checks for an overflow based on the type of `res`.
- The two operands in a binary operation might undergo promotion before the operation occurs. See also “Code Prover Assumptions About Implicit Data Type Conversions” (Polyspace Code Prover).

The exact storage allocation for different data types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer overflows on signed integers result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {

    int var = INT_MAX;
    var++;
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

#### Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

### Tainted sign change conversion

#### Issue

This issue occurs when values from unsecure sources are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
memmove
strncmp
strncpy
```

```
calloc
malloc
memalign
```

**Risk**

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

**Fix**

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

**Extend Checker**

By default, Polyspace assumes that data from external sources are tainted. See "Sources of Tainting in a Polyspace Analysis". To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

**Example — Set Memory Value with Size Argument**

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size); //Noncompliant
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

**Correction — Check Value of `size`**

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

### Unsigned integer constant overflow

#### Issue

This issue occurs in the following cases:

- You assign a compile-time constant to an unsigned integer variable whose data type cannot accommodate the value.
- You use an `enum` value that cannot be accommodated by the underlying type of the `enum` (and the underlying type is unsigned).

An `n`-bit unsigned integer holds values in the range [0, $2^n$-1]. For instance, `c` is an 8-bit unsigned `char` variable that cannot hold the value 256.

```
unsigned char c = 256;
```

This defect checker depends on the following options:

- To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.
- To determine the underlying types of enumerations, Bug Finder uses your specification for `Enum type definition (-enum-type-definition)`.

You do not see the defect in these situations:

- Creation of new constants from `const` variables (for specific compilers only).

  Different compilers might define compile-time constants differently. In the following code, `c+1u` is considered a compile time-constant by GCC compilers, but not by the standard C compiler:

  ```
  const uint16_t c = 0xffffu;
  uint16_t y = c + 1u;
  ```

  Whether you see a violation of this check on `y` might depend on your compiler.
- Bitwise `NOT` operation.

  Polyspace does not raise this violation when you perform a bitwise `NOT` operation.

#### Risk

The C standard states that overflowing unsigned integers must be wrapped around (see, for instance, the C11 standard, section 6.2.5). However, the wrap-around behavior can be unintended and cause unexpected results.

#### Fix

Check if the constant value is what you intended. If the value is correct, use a wider data type for the variable.

#### Example — Overflows from Assignments

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned short c2 = MAX_UNSIGNED_SHORT + 1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow.

#### Correction — Use Wider Data Type

One possible correction is to use a wider data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned short c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned int c2 = MAX_UNSIGNED_SHORT + 1;
}
```

### Unsigned integer conversion overflow

#### Issue

This issue occurs when converting an unsigned integer to a smaller unsigned integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer conversion overflows result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

#### Example — Converting from int to char

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum;
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned long)unum;
}
```

### Unsigned integer overflow

#### Issue

This issue occurs when an operation on unsigned integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

The C11 standard states that unsigned integer overflows result in wrap-around behavior. However, a wrap around behavior might not always be desirable. For instance, if the result of a computation is used as an array size and the computation overflows, the array size is much smaller than expected.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling. In the error handling code, you can override the default wrap-around behavior for overflows and implement saturation behavior, for instance.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Add One to Maximum Unsigned Integer

```
#include <limits.h>

unsigned int plusplus(void) {

    unsigned uvar = UINT_MAX;
    uvar++;
    return uvar;
}
```

In the third statement of this function, the variable `uvar` is increased by 1. However, the value of `uvar` is the maximum unsigned integer value, so 1 plus the maximum integer value cannot be represented by an `unsigned int`. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `uvar` is reduced by modulo `UINT_MAX`. The result is `uvar = 1`.

**Correction — Different Storage Type**

One possible correction is to store the operation result in a larger data type. In this example, by returning an `unsigned long long` instead of an `unsigned int`, the overflow error is fixed.

```
#include <limits.h>

unsigned long long plusplus(void) {

    unsigned long long ullvar = UINT_MAX;
    ullvar++;
    return ullvar;
}
```

## Check Information

**Category:** Numeric Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-128

# CWE Rule 129

Improper Validation of Array Index

## Description

### Rule Description

*The product uses untrusted input when calculating or using an array index, but the product does not validate or incorrectly validates the index to ensure the index references a valid position within the array.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access with tainted index**
- **Invalid use of standard library string routine**
- **Pointer dereference with tainted offset**
- **Tainted division operand**
- **Tainted modulo operand**

## Examples

### Array access with tainted index

#### Issue

This issue occurs when you access an array by using an index that is obtained from unsecure sources and which has not been validated.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite — writing to memory before the beginning of the buffer.
- Buffer overflow — writing to memory after the end of a buffer.
- Over-reading a buffer — accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write operation create to problems in your program.

#### Fix

Before using the index to access the array, validate the index value to make sure that it is inside the array range.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of

Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Use Index to Return Buffer Value

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
  return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    return tab[num];//Noncompliant
}
```

In this example, the index `num` accesses the array `tab`. The index `num` is obtained from an unsecure source and the function `taintedarrayindex` does not check to see if `num` is inside the range of `tab`.

#### Correction — Check Range Before Use

One possible correction is to check that `num` is in range before using it.

```
#include <stdlib.h>
#include <stdio.h>
#define SIZE100 100
extern int tab[SIZE100];
static int tainted_int_source(void) {
    return strtol(getenv("INDEX"),NULL,10);
}
int taintedarrayindex(void) {
    int num = tainted_int_source();
    if (num >= 0 && num < SIZE100) {
        return tab[num];
    } else {
        return -1;
    }
}
```

### Invalid use of standard library string routine

#### Issue

This issue occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to strcpy. In some cases, you can use an alternative function to avoid the error. For instance, instead of strcpy, you can use strncpy to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string text is larger in size than gbuffer. Therefore, the function strcpy cannot copy text into gbuffer.

#### Correction — Use Valid Arguments

One possible correction is to declare the destination string gbuffer with equal or larger size than the source string text.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Pointer dereference with tainted offset

#### Issue

This issue occurs when a pointer dereference uses an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

#### Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Dereference Pointer Array

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset];//Noncompliant
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

#### Correction — Check Index Before Dereference

One possible correction is to validate the value of `offset`. Continue with the pointer dereferencing only if `offset` is inside the valid range.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

### Tainted division operand

#### Issue

This issue occurs when one or both integer operands in a division operation comes from unsecure sources.

#### Risk

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

#### Fix

Before performing the division, validate the values of the operands. Check for denominators of `0` or `-1`, and numerators of the minimum integer value.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See "Sources of Tainting in a Polyspace Analysis". To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Division of Function Arguments

```
#include <limits.h>
#include <stdio.h>

extern void print_int(int);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    int r =  num/denum; //Noncompliant
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

#### Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include <limits.h>
#include <stdio.h>

extern void print_long(long);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    long res= 0;
    if (denum!=0 && !(num==INT_MIN && denum==-1)) {
        res = num/denum;
    }
    print_long(res);
    return res;
}
```

### Tainted modulo operand

#### Issue

This issue occurs when one or both integer operands in a remainder operation (%) comes from unsecure sources.

#### Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.

- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

#### Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of `0` and `-1`. Check both operands for negative values.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Modulo with User Input

```
#include <stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem =  128%userden; //Noncompliant
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using a user input. The input is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

#### Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
#include<stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem = 0;
    if (userden > 0 ) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

## Check Information

**Category:** Data Validation Issues

## Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-129

# CWE Rule 130

Improper Handling of Length Parameter Inconsistency

## Description

### Rule Description

*The software parses a formatted message or structure, but it does not handle or incorrectly handles a length field that is inconsistent with the actual length of the associated data.*

### Polyspace Implementation

The rule checker checks for **Mismatch between data length and size**.

## Examples

### Mismatch between data length and size

**Issue**

This issue occurs when you do not check the length argument and data buffer argument of memory copying functions such as `memcpy`, `memset`, or `memmove`, to protect against buffer overflows.

**Risk**

If an attacker can manipulate the data buffer or length argument, the attacker can cause buffer overflow by making the actual data size smaller than the length.

This mismatch in length allows the attacker to copy memory past the data buffer to a new location. If the extra memory contains sensitive information, the attacker can now access that data.

This defect is similar to the SSL Heartbleed bug.

**Fix**

When copying or manipulating memory, compute the length argument directly from the data so that the sizes match.

**Example — Copy Buffer of Data**

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;
    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    memcpy(&(beta.data[num]), os->data + 2, length);

    return(1);
}
```

This function copies the buffer `alpha` into a buffer `beta`. However, the `length` variable is not related to `data+2`.

#### Correction — Check Buffer Length

One possible correction is to check the length of your buffer against the maximum value minus 2. This check ensures that you have enough space to copy the data to the `beta` structure.

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    if (length<(os->max -2)) {
        memcpy(&(beta.data[num]), os->data + 2, length);
    }

    return(1);

}
```

## Check Information

**Category:** Data Processing Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-130

# CWE Rule 131

Incorrect Calculation of Buffer Size

## Description

### Rule Description

*The software does not correctly calculate the size to be used when allocating a buffer, which could lead to a buffer overflow.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access out of bounds**
- **Memory allocation with tainted size**
- **Pointer access out of bounds**
- **Tainted sign change conversion**
- **Tainted size of variable length array**
- **Wrong type used in sizeof**

## Examples

### Array access out of bounds

#### Issue

This issue occurs when an array index falls outside the range `[0...array_size-1]` during array access.

#### Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Extend Checker**

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

**Example — Array Access Out of Bounds Error**

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

**Correction — Keep Array Index Within Array Bounds**

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

### Memory allocation with tainted size

#### Issue

This issue occurs when a memory allocation function, such as `calloc` or `malloc`, uses a size argument from an unsecure source.

#### Risk

Uncontrolled memory allocation can cause your program to request too much system memory. This consequence can lead to a crash due to an out-of-memory condition, or assigning too many resources.

#### Fix

Before allocating memory, check the value of your arguments to check that they do not exceed the bounds.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Allocate Memory Using Input From User

```
#include<stdio.h>
#include <stdlib.h>

int* bug_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = (int*)malloc(size);//Noncompliant
    return p;
}
```

In this example, `malloc` allocates `size` bytes of memory for the pointer `p`. The variable `size` comes from the user of the program. Its value is not checked, and it could be larger than the amount of available memory. If `size` is larger than the number of available bytes, your program could crash.

#### Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the `malloc` operation. This example checks to see if `size` is positive and less than the maximum size.

```
#include<stdio.h>
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int* corrected_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = NULL;
    if (size>0 && size<SIZE128) {          /* Fix: Check entry range before use */
        p = (int*)malloc((unsigned int)size);
    }
    return p;
}
```

### Pointer access out of bounds

#### Issue

This issue occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

#### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

#### Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

### Tainted sign change conversion

#### Issue

This issue occurs when values from unsecure sources are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
memmove
strncmp
strncpy
calloc
malloc
memalign
```

**Risk**

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

**Fix**

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

**Extend Checker**

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

**Example — Set Memory Value with Size Argument**

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size); //Noncompliant
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

#### Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

### Tainted size of variable length array

#### Issue

This issue occurs when the size of a variable length array (VLA) is obtained from an unsecure source.

#### Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

#### Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Example — User Input Argument Used as Size of VLA

```
#include<stdio.h>
#inclule<stdlib.h>
#define LIM 40
```

```
long squaredSum(int size) {

    int tabvla[size];
    long res = 0;
    for (int i=0 ; i<LIM-1 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

#### Correction — Check VLA Size

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 0 and less than 40, before creating the VLA

```
#include <stdio.h>
#include <stdlib.h>
#define LIM 40

long squaredSum(int size) {
    long res = 0;
    if (size>0 && size<LIM){
        int tabvla[size];
        for (int i=0 ; i<size || i<LIM-1 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }else{
        res = -1;
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

### Wrong type used in sizeof

#### Issue

This issue occurs when both of the following conditions hold:

1 You assign the address of a block of memory to a pointer, or transfer data between two blocks of memory. The assignment or copy uses the `sizeof` operator.

   For instance, you initialize a pointer using `malloc(sizeof(`*`type`*`))` or copy data between two addresses using `memcpy(`*`destination_ptr`*`, `*`source_ptr`*`, sizeof(`*`type`*`))`.

2 You use an incorrect type as argument of the `sizeof` operator. For instance:

   - You might be using the pointer type instead of the type that the pointer points to. For example, to initialize a *`type`*`*` pointer, you might be using `malloc(sizeof(`*`type`*`*))` instead of `malloc(sizeof(`*`type`*`))`.
   - You might be using a completely unrelated type as `sizeof` argument. For example, to initialize a *`type`*`*` pointer, you might be using `malloc(sizeof(`*`anotherType`*`))`.

#### Risk

Irrespective of what *`type`* stands for, the expression `sizeof(`*`type`*`*)` always returns a fixed size. The size returned is the pointer size on your platform in bytes. The appearance of `sizeof(`*`type`*`*)` often indicates an unintended usage. The error can cause allocation of a memory block that is much smaller than what you need and lead to weaknesses such as buffer overflows.

For instance, assume that `structType` is a structure with ten `int` variables. If you initialize a `structType*` pointer using `malloc(sizeof(structType*))` on a 32-bit platform, the pointer is assigned a memory block of four bytes. However, to be allocated completely for one `structType` variable, the `structType*` pointer must point to a memory block of `sizeof(structType) = 10 * sizeof(int)` bytes. The required size is much greater than the actual allocated size of four bytes.

#### Fix

To initialize a *`type`*`*` pointer, replace `sizeof(`*`type`*`*)` in your pointer initialization expression with `sizeof(`*`type`*`)`.

#### Example — Allocate a Char Array With sizeof

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char*) * 5);
    free(str);

}
```

In this example, memory is allocated for the character pointer `str` using a `malloc` of five char pointers. However, `str` is a pointer to a character, not a pointer to a character pointer. Therefore the `sizeof` argument, `char*`, is incorrect.

#### Correction — Match Pointer Type to `sizeof` Argument

One possible correction is to match the argument to the pointer type. In this example, `str` is a character pointer, therefore the argument must also be a character.

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;
    str = (char*)malloc(sizeof(char) * 5);
    free(str);

}
```

## Check Information

**Category:** Memory Buffer Errors

# Version History

**Introduced in R2023a**

## See Also

Check CWE (-cwe)

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-131

# CWE Rule 134

Use of Externally-Controlled Format String

## Description

### Rule Description

*The software uses a function that accepts a format string as an argument, but the format string originates from an external source.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Format string specifiers and arguments mismatch**
- **Tainted string format**

## Examples

### Format string specifiers and arguments mismatch

#### Issue

This issue occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

#### Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

#### Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
const char *message = "License not available";
int err_number = ;-4
printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

In cases where integer promotion modifies the perceived data type of an argument, the analysis result shows both the original type and the type after promotion. The format specifier has to match the type after integer promotion.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.

- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example — Printing a Float**

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

**Correction — Use an Unsigned Long Format Specifier**

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

**Correction — Use an Integer Argument**

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

### Tainted string format

**Issue**

This issue occurs when `printf`-style functions use a format specifier constructed from unsecure sources.

**Risk**

If you use externally controlled elements to format a string, you can cause buffer overflow or data-representation problems. An attacker can use these string formatting elements to view the contents of a stack using `%x` or write to a stack using `%n`.

#### Fix

Pass a static string to format string functions. This fix ensures that an external actor cannot control the string.

Another possible fix is to allow only the expected number of arguments. If possible, use functions that do not support the vulnerable `%n` operator in format strings.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See "Sources of Tainting in a Polyspace Analysis". To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Get Elements from User Input

```
#include <stdio.h>
#include <unistd.h>
#define MAX 40
void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf(userstr);//Noncompliant
}
```

This example prints the input argument `userstr`. The string is unknown. If it contains elements such as `%`, `printf` can interpret `userstr` as a string format instead of a string, causing your program to crash.

#### Correction — Print as String

One possible correction is to print `userstr` explicitly as a string so that there is no ambiguity.

```
#include "stdio.h"
#include <unistd.h>
#define MAX 40

void taintedstringformat(void) {
    char userstr[MAX];
    read(0,userstr,MAX);
    printf("%.20s", userstr);
}
```

## Check Information

**Category:** String Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

#### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

CWE-134

# CWE Rule 135

Incorrect Calculation of Multi-Byte String Length

## Description

### Rule Description

*The software does not correctly calculate the length of strings that can contain wide or multi-byte characters.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Destination buffer overflow in string manipulation**
- **Misuse of narrow or wide character string**
- **Unreliable cast of pointer**

## Examples

### Destination buffer overflow in string manipulation

#### Issue

This issue occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

**Example — Buffer Overflow in sprintf Use**

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

**Correction — Use `snprintf` Instead of `sprintf`**

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Misuse of narrow or wide character string

**Issue**

This issue occurs when you pass a narrow character string to a wide string function, or a wide character string to a narrow string function.

**Misuse of narrow or wide character string** raises no defect on operating systems where narrow and wide character strings have the same size.

**Risk**

Using a narrow character string with a wide string function, or vice versa, can result in unexpected or undefined behavior.

If you pass a wide character string to a narrow string function, you can encounter these issues:

- Data truncation. If the string contains null bytes, a copy operation using `strncpy()` can terminate early.
- Incorrect string length. `strlen()` returns the number of characters of a string up to the first null byte. A wide string can have additional characters after its first null byte.

If you pass a narrow character string to a wide string function, you can encounter this issue:

- Buffer overflow. In a copy operation using `wcsncpy()`, the destination string might have insufficient memory to store the result of the copy.

**Fix**

Use the narrow string functions with narrow character strings. Use the wide string functions with wide character strings.

#### Example — Passing Wide Character Strings to strncpy()

```
#include <string.h>
#include <wchar.h>

void func(void)
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    strncpy(wide_str2, wide_str1, 10);
}
```

In this example, `strncpy()` copies 10 wide characters from `wide_strt1` to `wide_str2`. If `wide_str1` contains null bytes, the copy operation can end prematurely and truncate the wide character string.

#### Correction — Use `wcsncpy()` to Copy Wide Character Strings

One possible correction is to use `wcsncpy()` to copy `wide_str1` to `wide_str2`.

```
#include <string.h>
#include <wchar.h>

void func(void)
{
    wchar_t wide_str1[]  = L"0123456789";
    wchar_t wide_str2[] =  L"0000000000";
    wcsncpy(wide_str2, wide_str1, 10);
}
```

### Unreliable cast of pointer

#### Issue

This issue occurs when a pointer is implicitly cast to a data type different from its declaration type. Such an implicit casting can take place, for instance, when a pointer to data type `char` is assigned the address of an integer.

This defect applies only if the code language for the project is C.

#### Risk

Casting a pointer to data type different from its declaration type can result in issues such as buffer overflow. If the cast is implicit, it can indicate a coding error.

#### Fix

Avoid *implicit* cast of a pointer to a data type different from its declaration type.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.

- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example — Unreliable cast of pointer error

```
#include <string.h>

void Copy_Integer_To_String()
{
 int src[]={1,2,3,4,5,6,7,8,9,10};
 char buffer[]="Buffer_Text";
 strcpy(buffer,src);
 /* Defect: Implicit cast of (int*) to (char*) */
}
```

`src` is declared as an `int*` pointer. The `strcpy` statement, while copying to `buffer`, implicitly casts `src` to `char*`.

#### Correction — Avoid Pointer Cast

One possible correction is to declare the pointer `src` with the same data type as `buffer`.

```
#include <string.h>
 void Copy_Integer_To_String()
{
 /* Fix: Declare src with same type as buffer */
 char *src[10]={"1","2","3","4","5","6","7","8","9","10"};
 char *buffer[10];

 for(int i=0;i<10;i++)
   buffer[i]="Buffer_Text";

 for(int i=0;i<10;i++)
   buffer[i]= src[i];
 }
```

## Check Information

**Category:** String Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-135

# CWE Rule 170

Improper Null Termination

## Description

### Rule Description

*The software does not terminate or incorrectly terminates a string or array with a null character or equivalent terminator.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Invalid use of standard library string routine**
- **Missing null in string array**
- **Misuse of readlink()**
- **Tainted NULL or non-null-terminated string**

## Examples

### Invalid use of standard library string routine

#### Issue

This issue occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.

- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example — Invalid Use of Standard Library String Routine Error**

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

**Correction — Use Valid Arguments**

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Missing null in string array

**Issue**

This issue occurs when a string does not have enough space to terminate with a null character `'\0'`.

This defect applies only for projects in C.

**Risk**

A buffer overflow can occur if you copy a string to an array without assuming the implicit null terminator.

**Fix**

If you initialize a character array with a literal, avoid specifying the array bounds.

```
char three[]  = "THREE";
```

The compiler automatically allocates space for a null terminator. In the preceding example, the compiler allocates sufficient space for five characters and a null terminator.

If the issue occurs after initialization, you might have to increase the size of the array by one to account for the null terminator.

In certain circumstances, you might want to initialize the character array with a sequence of characters instead of a string. In this situation, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example — Array size is too small**

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE";
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

**Correction — Increase Array Size**

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

**Correction — Change Initialization Method**

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

### Misuse of readlink()

**Issue**

This issue occurs when you pass a buffer size argument to `readlink()` that does not leave space for a null terminator in the buffer.

For instance:

```
ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf));
```

The third argument is exactly equal to the size of the second argument. For large enough symbolic links, this use of `readlink()` does not leave space to enter a null terminator.

**Risk**

The `readlink()` function copies the content of a symbolic link (first argument) to a buffer (second argument). However, the function does not append a null terminator to the copied content. After using `readlink()`, you must explicitly add a null terminator to the buffer.

If you fill the entire buffer when using `readlink`, you do not leave space for this null terminator.

**Fix**

When using the `readlink()` function, make sure that the third argument is one less than the buffer size.

Then, append a null terminator to the buffer. To determine where to add the null terminator, check the return value of `readlink()`. If the return value is -1, an error has occurred. Otherwise, the return value is the number of characters (bytes) copied.

**Example — Incorrect Size Argument of readlink**

```
#include <unistd.h>

#define SIZE1024 1024

extern void display_path(const char *);

void func() {
    char buf[SIZE1024];
    ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf));
    if (len > 0) {
        buf[len - 1] = '\0';
    }
    display_path(buf);
}
```

In this example, the third argument of `readlink` is exactly the size of the buffer (second argument). If the first argument is long enough, this use of `readlink` does not leave space for the null terminator.

Also, if no characters are copied, the return value of `readlink` is 0. The following statement leads to a buffer underflow when `len` is 0.

```
buf[len - 1] = '\0';
```

**Correction — Make Sure Size Argument is One Less Than Buffer Size**

One possible correction is to make sure that the third argument of `readlink` is one less than size of the second argument.

The following corrected code also accounts for `readlink` returning 0.

```
#include <stdlib.h>
#include <unistd.h>
```

```
#define fatal_error() abort()
#define SIZE1024 1024

extern void display_path(const char *);

void func() {
    char buf[SIZE1024];
    ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf) - 1);
    if (len != -1) {
        buf[len] = '\0';
        display_path(buf);
    }
    else {
        /* Handle error */
        fatal_error();
    }
}
```

### Tainted NULL or non-null-terminated string

#### Issue

This issue occurs when strings from unsecure sources are used in string manipulation routines that implicitly dereference the string buffer, for instance, `strcpy` or `sprintf`.

**Tainted NULL or non-null-terminated string** raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

#### Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

#### Fix

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Getting String from Input

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));//Noncompliant
    print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

**Correction — Validate the Data**

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // Noncompliant-TAINTED_STRING only flagged here
        // - string is not null
        // - string has a positive and limited size
        // - TAINTED_STRING on strlen used as a firewall
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

**Correction — Validate the Data**

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

## Check Information

**Category:** Data Neutralization Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-170

# CWE Rule 188

Reliance on Data/Memory Layout

## Description

### Rule Description

*The software makes invalid assumptions about how protocol data or memory is organized at a lower level, resulting in unintended program behavior.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Invalid assumptions about memory organization**
- **Memory comparison of padding data**
- **Missing byte reordering when transferring data**

## Examples

### Invalid assumptions about memory organization

#### Issue

This issue occurs when you compute the address of a variable in the stack by adding or subtracting from the address of another non-array variable.

#### Risk

When you compute the address of a variable in the stack by adding or subtracting from the address of another variable, you assume a certain memory organization. If your assumption is incorrect, accessing the computed address can be invalid.

#### Fix

Do not perform an access that relies on assumptions about memory organization.

#### Example — Reliance on Memory Organization

```
void func(void) {
    int var1 = 0x00000011, var2;
    *(&var1 + 1) = 0;
}
```

In this example, the programmer relies on the assumption that `&var1 + 1` provides the address of `var2`. Therefore, an **Invalid assumptions about memory organization** appears on the `+` operation. In addition, a **Pointer access out of bounds** error also appears on the dereference.

#### Correction — Do Not Rely on Memory Organization

One possible correction is not perform direct computation on addresses to access separately declared variables.

### Memory comparison of padding data

#### Issue

This issue occurs when you use the `memcmp` function to compare two structures as a whole. In the process, you compare meaningless data stored in the structure padding.

For instance:

```
struct structType {
    char member1;
    int member2;
    .
    .
};

structType var1;
structType var2;
.
.
if(memcmp(&var1,&var2,sizeof(var1)))
{...}
```

#### Risk

If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. For an example of padding, see `Higher Estimate of Size of Local Variables`.

The content of these extra padding bytes is meaningless. The C Standard allows the content of these bytes to be indeterminate, giving different compilers latitude to implement their own padding. If you perform a byte-by-byte comparison of structures with `memcmp`, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.

#### Fix

Instead of comparing two structures in one attempt, compare the structures member by member.

For efficient code, write a function that does the comparison member by member. Use this function for comparing two structures.

You can use `memcmp` for byte-by-byte comparison of structures only if you know that the structures do not contain padding. Typically, to prevent padding, you use specific attributes or pragmas such as `#pragma pack`. However, these attributes or pragmas are not supported by all compilers and make your code implementation-dependent. If your structures contain bit-fields, using these attributes or pragmas cannot prevent padding.

#### Example — Structures Compared with memcmp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
{

    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    if (0 == memcmp(left, right, sizeof(S_Padding)))
    {
        return 1;
    }
    else
        return 0;
}
```

In this example, `memcmp` compares byte-by-byte the two structures that `left` and `right` point to. Even if the values stored in the structure members are the same, the comparison can show an inequality if the meaningless values in the padding bytes are not the same.

**Correction — Compare Structures Member by Member**

One possible correction is to compare individual structure members.

**Note** You can compare entire arrays by using `memcmp`. All members of an array have the same data type. Padding bytes are not required to store arrays.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define fatal_error() abort()

typedef struct s_padding
{
    char c;
    int i;
    unsigned int bf1:1;
    unsigned int bf2:2;
    unsigned char buffer[20];
} S_Padding ;

/* Function that guarantees safe access to the input memory */
extern int trusted_memory_zone(void *ptr, size_t sz);

int func(const S_Padding *left, const S_Padding *right)
```

```
{
    if (!trusted_memory_zone((void *)left, sizeof(S_Padding)) ||
        !trusted_memory_zone((void *)right, sizeof(S_Padding))) {
        fatal_error();
    }

    return ((left->c == right->c) &&
            (left->i == right->i) &&
            (left->bf1 == right->bf1) &&
            (left->bf2 == right->bf2) &&
            (memcmp(left->buffer, right->buffer, 20) == 0));
}
```

### Missing byte reordering when transferring data

#### Issue

This issue occurs when you do not use a byte ordering function:

- Before sending data to a network socket.
- After receiving data from a network socket.

#### Risk

Some system architectures implement little endian byte ordering (least significant byte first), and other systems implement big endian (most significant byte first). If the endianness of the sent data does not match the endianness of the receiving system, the value returned when reading the data is incorrect.

#### Fix

After receiving data from a socket, use a byte ordering function such as `ntohl()`. Before sending data to a socket, use a byte ordering function such as `htonl()` .

#### Example — Data Transferred Without Byte Reordering

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <byteswap.h>
#include <unistd.h>
#include <string.h>


unsigned int func(int sock, int server)
{
    unsigned int num;   /* assume int is 32-bits */
    if (server)
    {
        /* Server side */
        num = 0x17;
        /* Endianness of server host may not match endianness of network. */
        if (send(sock, (void *)&num, sizeof(num), 0) < (int)sizeof(num))
        {
            /* Handle error */
        }
        return 0;
    }
    else {
        /* Endianness of client host may not match endianness of network. */
        if (recv (sock, (void *)&num, sizeof(num), 0) < (int) sizeof(num))
        {
            /* Handle error */
        }

        /* Comparison may be inaccurate */
        if (num> 255)
        {
            return 255;
        }
        else
        {
            return num;
        }
    }
}
```

In this example, variable `num` is assigned hexadecimal value `0x17` and is sent over a network to the client from the server. If the server host is little endian and the network is big endian, `num` is transferred as `0x17000000`. The client then reads an incorrect value for `num` and compares it to a local numeric value.

**Correction — Use Byte Ordering Function**

Before sending `num` from the server host, use `htonl()` to convert from host to network byte ordering. Similarly, before reading `num` on the client host, use `ntohl()` to convert from network to host byte ordering.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <byteswap.h>
#include <unistd.h>
#include <string.h>

unsigned int func(int sock, int server)
{
    unsigned int num;   /* assume int is 32-bits */
    if (server)
    {
        /* Server side */
        num = 0x17;

        /* Convert to network byte order. */
        num = htonl(num);
        if (send(sock, (void *)&num, sizeof(num), 0) < (int)sizeof(num))
        {
            /* Handle error */
        }
        return 0;
    }
    else {
        if (recv (sock, (void *)&num, sizeof(num), 0) < (int) sizeof(num))
        {
            /* Handle error */
        }

        /* Convert to host byte order. */
        num = ntohl(num);
        if (num > 255)
        {
            return 255;
        }
        else
        {
            return num;
        }
    }
}
```

## Check Information

**Category:** Data Neutralization Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-188

# CWE Rule 191

Integer Underflow (Wrap or Wraparound)

## Description

### Rule Description

*The product subtracts one value from another, such that the result is less than the minimum allowable integer value, which produces a value that is not equal to the correct result.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Integer constant overflow**
- **Integer overflow**
- **Unsigned integer constant overflow**
- **Unsigned integer conversion overflow**
- **Unsigned integer overflow**

## Examples

### Integer constant overflow

#### Issue

This issue occurs in the following cases:

- You assign a compile-time integer constant to a signed integer variable whose data type cannot accommodate the value.

  See “Constant Overflows from Assignments” on page 7-17.
- You use an `enum` value that cannot be accommodated by the underlying type of the `enum` (and the underlying type is signed). For most C compilers, the default underlying type is `signed int` (based on the C standard).

  See “Constant Overflows from enum Values” on page 7-17.
- You perform a binary operation involving two integer constants that results in an overflow, that is, a value outside the range allowed by the data type that the operation uses. A binary operation with integer constants uses the `signed int` data type (unless you use modifiers such as `u` or `L`).

  See “Constant Overflows from Binary Operations” on page 7-18.

An `n`-bit signed integer holds values in the range [-$2^{n-1}$, $2^{n-1}$-1]. For instance, `c` is an 8-bit signed `char` variable that cannot hold the value 255.

```
signed char c = 255;
```

This defect checker depends on the following options:

- `Target processor type (-target)`: Determines the sizes of fundamental types.
- `Enum type definition (-enum-type-definition)`: Determines the underlying types of enumerations.
- `Compiler (-compiler)`: Impacts the interpretation of code.

You do not see the defect in these situations:

- Creation of new constants from `const` variables (for specific compilers only).

  Different compilers might define compile-time constants differently. In the following code, `c+1` is considered a compile-time constant by GCC compilers, but not by the standard C compiler:

  ```
  const int16_t c = 32767;
  int16_t y = c + 1;
  ```

  Whether you see a violation of this check on `y` might depend on your compiler.
- Bitwise `NOT` operation.

  Polyspace does not raise this violation when you perform a bitwise `NOT` operation.

#### Risk

The default behavior for constant overflows can vary between compilers and platforms. Retaining constant overflows can reduce the portability of your code.

Even if your compilers wraps around overflowing constants with a warning, the wrap-around behavior can be unintended and cause unexpected results.

#### Fix

Check if the constant value is what you intended. If the value is correct, use a different, possibly wider, data type for the variable.

#### Example — Constant Overflows from Assignments

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    char c1 = MAX_UNSIGNED_CHAR;
    char c2 = MAX_SIGNED_CHAR+1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow. To reproduce these defects, use a `Target processor type (-target)` where `char` is signed by default.

#### Correction — Use Different Data Type

One possible correction is to use a different data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_SIGNED_CHAR 127

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR;
    unsigned char c2 = MAX_SIGNED_CHAR+1;
}
```

#### Example — Constant Overflows from enum Values

```
enum {
  a=0x7fffffff,
  b,
  c=0xffffffff
} MyEnumA;
```

In this example, the `enum` has an underlying type `int`. The `int` type can accommodate values in the range [-$2^{31}$, $2^{31}$-1]. However, the value of the enumerator `b` is 0x80000000 or $2^{31}$ (one more than the previous value `a`). This value falls outside the allowed range of `int`.

The value of `c`, that is 0xffffffff or $2^{32}$-1, is even larger and also causes an overflow.

To see this defect:

- Specify `Compiler (-compiler)` as `generic`.
- Specify `Source code language (-lang)` as `c`.

#### Example — Constant Overflows from Binary Operations

```
const unsigned int K_ATM_Label_Ram_init_value [] = {
     0x06 | ( 3 << 29) ,
     0x80 | ( 9 << 29) ,
     ( 2 << 31 )        ,
};
```

In this example, two of the shift operations result in values that cannot be accommodated by the `signed int` data type. The `signed int` data type can accommodate values in the range [-$2^{31}$, $2^{31}$-1]. The operation:

- `9 << 29` results in the value $2^{32}$+536870912.
- `2 << 31` results in the value $2^{32}$.

Note that even though the result is assigned to an `unsigned int` variable, the overflow detection uses the underlying type of the binary operation, that is, `signed int`.

### Integer overflow

#### Issue

This issue occurs when an operation on integer variables results in values that cannot be represented by the data type that the operation uses. This data type depends on the operand types and determines the number of bytes allocated for storing the result, thus constraining the range of allowed values.

Note that:

- The data type used to determine an overflow is based on the operand data types. If you then assign the result of an operation to another variable, a different checker, `Integer conversion overflow`, determines if the value assigned also overflows the variable assigned to. For instance, in an operation such as:

  ```
  res = x + y;
  ```

This checker checks for an overflow based on the types of `x` and `y`, and not on the type of `res`. The checker for integer conversion overflows then checks for an overflow based on the type of `res`.
- The two operands in a binary operation might undergo promotion before the operation occurs. See also “Code Prover Assumptions About Implicit Data Type Conversions” (Polyspace Code Prover).

The exact storage allocation for different data types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer overflows on signed integers result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

To avoid overflows in general, try one of these techniques:

- Keep integer variable values restricted to within half the range of signed integers.
- In operations that might overflow, check for conditions that can lead to the overflow and implement wrap around or saturation behavior depending on how the result of the operation is used. The result then becomes predictable and can be safely used in subsequent computations.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Addition of Maximum Integer

```
#include <limits.h>

int plusplus(void) {
    int var = INT_MAX;
    var++;
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

**Correction — Different Storage Type**

One possible correction is to change data types. Store the result of the operation in a larger data type (Note that on a 32-bit machine, `int` and `long` has the same size). In this example, on a 32-bit machine, by returning a `long long` instead of an `int`, the overflow error is fixed.

```
#include <limits.h>

long long plusplus(void) {

    long long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

### Unsigned integer constant overflow

**Issue**

This issue occurs in the following cases:

- You assign a compile-time constant to an unsigned integer variable whose data type cannot accommodate the value.
- You use an `enum` value that cannot be accommodated by the underlying type of the `enum` (and the underlying type is unsigned).

An `n`-bit unsigned integer holds values in the range [0, $2^n$-1]. For instance, `c` is an 8-bit unsigned `char` variable that cannot hold the value 256.

```
unsigned char c = 256;
```

This defect checker depends on the following options:

- To determine the sizes of fundamental types, Bug Finder uses your specification for `Target processor type (-target)`.
- To determine the underlying types of enumerations, Bug Finder uses your specification for `Enum type definition (-enum-type-definition)`.

You do not see the defect in these situations:

- Creation of new constants from `const` variables (for specific compilers only).

  Different compilers might define compile-time constants differently. In the following code, `c+1u` is considered a compile time-constant by GCC compilers, but not by the standard C compiler:

  ```
  const uint16_t c = 0xffffu;
  uint16_t y = c + 1u;
  ```

  Whether you see a violation of this check on `y` might depend on your compiler.

- Bitwise NOT operation.

  Polyspace does not raise this violation when you perform a bitwise NOT operation.

#### Risk

The C standard states that overflowing unsigned integers must be wrapped around (see, for instance, the C11 standard, section 6.2.5). However, the wrap-around behavior can be unintended and cause unexpected results.

#### Fix

Check if the constant value is what you intended. If the value is correct, use a wider data type for the variable.

#### Example — Overflows from Assignments

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned char c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned short c2 = MAX_UNSIGNED_SHORT + 1;
}
```

In this example, the defect appears on the macros because at least one use of the macro causes an overflow.

#### Correction — Use Wider Data Type

One possible correction is to use a wider data type for the variables that overflow.

```
#define MAX_UNSIGNED_CHAR 255
#define MAX_UNSIGNED_SHORT 65535

void main() {
    unsigned short c1 = MAX_UNSIGNED_CHAR + 1;
    unsigned int c2 = MAX_UNSIGNED_SHORT + 1;
}
```

### Unsigned integer conversion overflow

#### Issue

This issue occurs when converting an unsigned integer to a smaller unsigned integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer conversion overflows result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Converting from int to char

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum;
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned long)unum;
}
```

### Unsigned integer overflow

#### Issue

This issue occurs when an operation on unsigned integer variables can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

The C11 standard states that unsigned integer overflows result in wrap-around behavior. However, a wrap around behavior might not always be desirable. For instance, if the result of a computation is used as an array size and the computation overflows, the array size is much smaller than expected.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling. In the error handling code, you can override the default wrap-around behavior for overflows and implement saturation behavior, for instance.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Add One to Maximum Unsigned Integer

```
#include <limits.h>

unsigned int plusplus(void) {
    unsigned uvar = UINT_MAX;
    uvar++;
    return uvar;
}
```

In the third statement of this function, the variable `uvar` is increased by 1. However, the value of `uvar` is the maximum unsigned integer value, so 1 plus the maximum integer value cannot be represented by an `unsigned int`. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `uvar` is reduced by modulo `UINT_MAX`. The result is `uvar = 1`.

#### Correction — Different Storage Type

One possible correction is to store the operation result in a larger data type. In this example, by returning an `unsigned long long` instead of an `unsigned int`, the overflow error is fixed.

```
#include <limits.h>

unsigned long long plusplus(void) {

    unsigned long long ullvar = UINT_MAX;
    ullvar++;
    return ullvar;
}
```

## Check Information

**Category:** Numeric Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-191

# CWE Rule 192

Integer Coercion Error

## Description

### Rule Description

*Integer coercion refers to a set of flaws pertaining to the type casting, extension, or truncation of primitive data types.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Integer conversion overflow**
- **Sign change integer conversion overflow**
- **Tainted sign change conversion**
- **Unsigned integer conversion overflow**

## Examples

### Integer conversion overflow

#### Issue

This issue occurs when converting an integer to a smaller integer type. If the variable does not have enough bytes to represent the original value, the conversion overflows. For instance, if you perform a comparison between implementation-defined type `time_t` and a signed integer, Polyspace reports a violation because `time_t` might be implemented as an unsigned integer.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer conversion overflows result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

#### Example — Converting from int to char

```
char convert(void) {

    int num = 1000000;

    return (char)num;
}
```

In the return statement, the integer variable `num` is converted to a `char`. However, an 8-bit or 16-bit character cannot represent 1000000 because it requires at least 20 bits. So the conversion operation overflows.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number.

```
long convert(void) {

    int num = 1000000;

    return (long)num;
}
```

### Sign change integer conversion overflow

#### Issue

This issue occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Convert from unsigned char to char

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

#### Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

### Tainted sign change conversion

#### Issue

This issue occurs when values from unsecure sources are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
```

```
memmove
strncmp
strncpy
calloc
malloc
memalign
```

**Risk**

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

**Fix**

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

**Extend Checker**

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

**Example — Set Memory Value with Size Argument**

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size); //Noncompliant
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

#### Correction — Check Value of size

One possible correction is to check if size is inside the valid range. This correction checks if size is greater than zero and less than the buffer size before calling memset.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

### Unsigned integer conversion overflow

#### Issue

This issue occurs when converting an unsigned integer to a smaller unsigned integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See Target processor type (-target).

#### Risk

Integer conversion overflows result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

#### Example — Converting from int to char

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum;
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned long)unum;
}
```

## Check Information

**Category:** Numeric Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-192

# CWE Rule 194

Unexpected Sign Extension

## Description

### Rule Description

*The software performs an operation on a number that causes it to be sign extended when it is transformed into a larger data type. When the original number is negative, this can produce unexpected values that lead to resultant weaknesses.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Sign change integer conversion overflow**
- **Tainted sign change conversion**

## Examples

### Sign change integer conversion overflow

#### Issue

This issue occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

#### Example — Convert from unsigned char to char

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

#### Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

### Tainted sign change conversion

#### Issue

This issue occurs when values from unsecure sources are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
memmove
strncmp
strncpy
calloc
malloc
memalign
```

#### Risk

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

#### Fix

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size); //Noncompliant
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

#### Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

Check CWE (-cwe)

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-194

# CWE Rule 195

Signed to Unsigned Conversion Error

## Description

### Rule Description

*The software uses a signed primitive and performs a cast to an unsigned primitive, which can produce an unexpected value if the value of the signed primitive can not be represented using an unsigned primitive.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Sign change integer conversion overflow**
- **Tainted sign change conversion**

## Examples

### Sign change integer conversion overflow

#### Issue

This issue occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Convert from unsigned char to char

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

#### Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

### Tainted sign change conversion

#### Issue

This issue occurs when values from unsecure sources are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
memmove
strncmp
strncpy
calloc
malloc
memalign
```

#### Risk

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

#### Fix

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size); //Noncompliant
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

#### Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-195

# CWE Rule 196

Unsigned to Signed Conversion Error

## Description

### Rule Description

*The software uses an unsigned primitive and performs a cast to a signed primitive, which can produce an unexpected value if the value of the unsigned primitive can not be represented using a signed primitive.*

### Polyspace Implementation

The rule checker checks for **Sign change integer conversion overflow**.

## Examples

### Sign change integer conversion overflow

#### Issue

This issue occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Convert from unsigned char to char

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

#### Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-196

# CWE Rule 197

Numeric Truncation Error

## Description

### Rule Description

*Truncation errors occur when a primitive is cast to a primitive of a smaller size and data is lost in the conversion.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Call to memset family with unintended value**
- **Float conversion overflow**
- **Integer conversion overflow**
- **Sign change integer conversion overflow**
- **Tainted sign change conversion**
- **Unsigned integer conversion overflow**

## Examples

### Call to memset family with unintended value

#### Issue

This issue occurs when Polyspace Bug Finder detects a use of the `memset` or `wmemset` function with possibly incorrect arguments.

`void *memset (void *ptr, int value, size_t num)` fills the first `num` bytes of the memory block that `ptr` points to with the specified `value`. If the argument `value` is incorrect, the memory block is initialized with an unintended value.

The unintended initialization can occur in the following cases.

| Issue | Risk | Possible Fix |
|---|---|---|
| The second argument is `'0'` instead of `0` or `'\0'`. | The ASCII value of character `'0'` is `48` (decimal), `0x30` (hexadecimal), `069` (octal) but not `0` (or `'\0'`) . | If you want to initialize with `'0'`, use one of the ASCII values. Otherwise, use `0` or `'\0'`. |
| The second and third arguments are probably reversed. For instance, the third argument is a literal and the second argument is not a literal. | If the order is reversed, a memory block of unintended size is initialized with incorrect arguments. | Reverse the order of the arguments. |

| Issue | Risk | Possible Fix |
|---|---|---|
| The second argument cannot be represented in a byte. | If the second argument cannot be represented in a byte, and you expect each byte of a memory block to be filled with that argument, the initialization does not occur as intended. | Apply a bit mask to the argument to produce a wrapped or truncated result that can be represented in a byte. When you apply a bit mask, make sure that it produces an expected result.<br>For instance, replace `memset(a, -13, sizeof(a))` with `memset(a, (-13) & 0xFF, sizeof(a))`. |

**Fix**

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example — Value Cannot Be Represented in a Byte**

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE];
    int c = -2;
    memset(buf, (char)c, sizeof(buf));
}
```

In this example, `(char)c` cannot be represented in a byte.

**Correction — Apply Cast**

One possible correction is to apply a cast so that the result can be represented in a byte. Check that the result of the cast is an acceptable initialization value. In this correction, Polyspace does not raise this defect. The cast from signed `int` to `unsigned char` is contrary to best practices and Polyspace raises the defect `Sign change integer conversion overflow`.

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE   ];
    int c = -2;
    memset(buf, (unsigned char)c, sizeof(buf));// Might Overflow
}
```

#### Correction — Avoid Using `memset`

One possible correction is to reserve the use of `memset` only for setting or clearing all bits in a buffer. For instance, in this code, `memset` is called to clear the bits of the character array `buf`.

```
#include <string.h>

#define SIZE 32
void func(void) {
    char buf[SIZE   ];
    int c = -2;
    memset(buf, 0, sizeof(buf));//Compliant
    /* After clearing buf, use it in operations*/
}
```

### Float conversion overflow

#### Issue

This issue occurs when converting a floating point number to a smaller floating point data type. If the variable does not have enough memory to represent the original number, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Overflows can result in unpredictable values from computations. The result can be infinity or the maximum finite value depending on the rounding mode used in the implementation. If you use the result of an overflowing conversion in subsequent computations and do not account for the overflow, you can see unexpected results.

#### Fix

The fix depends on the root cause of the defect. Often the result details show a sequence of events that led to the defect. Use this event list to determine how the variable being converted acquires its current value You can implement the fix on any event in the sequence. If the result details do not show the event history, you can trace back using right-click options in the source code and see previous related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller floating point types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.

- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and `NaNs`. Operations that results in infinities and `NaNs` might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats).`

**Extend Checker**

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

**Example — Converting from double to float**

```
float convert(void) {

    double diam = 1e100;
    return (float)diam;
}
```

In the return statement, the variable `diam` of type double (64 bits) is converted to a variable of type float (32 bits). However, the value 1^100 requires more than 32 bits to be precisely represented.

### Integer conversion overflow

**Issue**

This issue occurs when converting an integer to a smaller integer type. If the variable does not have enough bytes to represent the original value, the conversion overflows. For instance, if you perform a comparison between implementation-defined type `time_t` and a signed integer, Polyspace reports a violation because `time_t` might be implemented as an unsigned integer.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

**Risk**

Integer conversion overflows result in undefined behavior.

**Fix**

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface" or "Interpret Bug Finder Results in Polyspace Access Web Interface".

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

#### Example — Converting from int to char

```
char convert(void) {

    int num = 1000000;

    return (char)num;
}
```

In the return statement, the integer variable `num` is converted to a `char`. However, an 8-bit or 16-bit character cannot represent 1000000 because it requires at least 20 bits. So the conversion operation overflows.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number.

```
long convert(void) {

    int num = 1000000;

    return (long)num;
}
```

### Sign change integer conversion overflow

#### Issue

This issue occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface" or "Interpret Bug Finder Results in Polyspace Access Web Interface".

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

#### Example — Convert from unsigned char to char

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

#### Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

### Tainted sign change conversion

#### Issue

This issue occurs when values from unsecure sources are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
memmove
strncmp
strncpy
calloc
malloc
memalign
```

#### Risk

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

#### Fix

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See "Sources of Tainting in a Polyspace Analysis". To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size); //Noncompliant
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

#### Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

```
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

### Unsigned integer conversion overflow

#### Issue

This issue occurs when converting an unsigned integer to a smaller unsigned integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer conversion overflows result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Converting from int to char

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum;
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned long)unum;
}
```

## Check Information

**Category:** Numeric Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-197

# CWE Rule 242

Use of Inherently Dangerous Function

## Description

### Rule Description

*The program calls a function that can never be guaranteed to work safely.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Use of dangerous standard function**
- **Use of obsolete standard function**

## Examples

### Use of dangerous standard function

#### Issue

This issue occurs when your code uses standard functions that write data to a buffer in a way that can result in buffer overflows.

The following table lists dangerous standard functions, the risks of using each function, and what function to use instead. The checker flags:

- Any use of an inherently dangerous function.
- An use of a possibly dangerous function only if the size of the buffer to which data is written can be determined at compile time. The checker does not flag an use of such a function with a dynamically allocated buffer.

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `gets` | Inherently dangerous — You cannot control the length of input from the console. | `fgets` |
| `std::cin::operator>>` and `std::wcin::operator>>` | Inherently dangerous — You cannot control the length of input from the console. | Preface calls to `cin` by `cin.width` to control the input length. This method can result in truncated input.<br>To avoid potential buffer overflow and truncated input, use `std::string` objects as destinations for >> operator. |

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `strcpy` | Possibly dangerous — If the size of the destination buffer is too small to accommodate the source buffer and a null terminator, a buffer overflow might occur. | Use the function `strlen()` to determine the size of the source buffer, and allocate sufficient memory so that the destination buffer can accommodate the source buffer and a null terminator. Instead of `strcpy`, use the function `strncpy`. |
| `stpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `stpncpy` |
| `lstrcpy` or `StrCpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy` |
| `strcat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `strncat`, `strlcat`, or `strcat_s` |
| `lstrcat` or `StrCat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat` |
| `wcpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcpncpy` |
| `wcscat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `wcsncat`, `wcslcat`, or `wcncat_s` |
| `wcscpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcsncpy` |
| `sprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `snprintf` |
| `vsprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `vsnprintf` |

#### Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

#### Fix

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.

- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Example — Using sprintf

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

### Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

### Use of obsolete standard function

#### Issue

This issue occurs when you use standard function routines that are considered legacy, removed, deprecated, or obsolete by C/C++ coding standards.

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| `asctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |
| `asctime_r` | Deprecated in POSIX.1-2008 | Implementation based on unsafe function `sprintf`. | `strftime` or `asctime_s` |
| `bcmp` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcmp` |
| `bcopy` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcpy` or `memmove` |
| `brk` and `sbrk` | Marked as legacy in SUSv2 and POSIX.1-2001. | | `malloc` |
| `bsd_signal` | Removed in POSIX.1-2008 | | `sigaction` |
| `bzero` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | `memset` |
| `ctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |
| `ctime_r` | Deprecated in POSIX.1-2008 | Implementation based on unsafe function `sprintf`. | `strftime` or `asctime_s` |
| `cuserid` | Removed in POSIX.1-2001. | Not reentrant. Precise functionality not standardized causing portability issues. | `getpwuid` |
| `ecvt` and `fcvt` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | Not reentrant | `snprintf` |
| `ecvt_r` and `fcvt_r` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | | `snprintf` |
| `ftime` | Removed in POSIX.1-2008 | | `time`, `gettimeofday`, `clock_gettime` |
| `gamma`, `gammaf`, `gammal` | Function not specified in any standard because of historical variations | Portability issues. | `tgamma`, `lgamma` |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| gcvt | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | snprintf |
| getcontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| getdtablesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_OPEN _MAX ) |
| gethostbyaddr | Removed in POSIX.1-2008 | Not reentrant | getaddrinfo |
| gethostbyname | Removed in POSIX.1-2008 | Not reentrant | getnameinfo |
| getpagesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_PAGE SIZE ) |
| getpass | Removed in POSIX.1-2001. | Not reentrant. | getpwuid |
| getw | Not present in POSIX.1-2001. | | fread |
| getwd | Marked legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | getcwd |
| index | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strchr |
| makecontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| memalign | Appears in SunOS 4.1.3. Not in 4.4 BSD or POSIX.1-2001 | | posix_memalign |
| mktemp | Removed in POSIX.1-2008. | Generated names are predictable and can cause a race condition. | mkstemp removes race risk |
| pthread_attr_ getstackaddr and pthread_attr_ setstackaddr | | Ambiguities in the specification of the stackaddr attribute cause portability issues | pthread_attr_ getstack and pthread_attr_ setstack |
| putw | Not present in POSIX.1-2001. | Portability issues. | fwrite |
| qecvt and qfcvt | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | snprintf |
| qecvt_r and qfcvt_r | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | snprintf |
| rand_r | Marked as obsolete in POSIX.1-2008 | | |
| re_comp | BSD API function | Portability issues | regcomp |
| re_exes | BSD API function | Portability issues | regexec |
| rindex | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strrchr |
| scalb | Removed in POSIX.1-2008 | | scalbln, scalblnf, or scalblnl |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| sigblock | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigmask | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigsetmask | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigstack | Interface is obsolete and not implemented on most platforms. | Portability issues. | sigaltstack |
| sigvec | 4.3BSD signal API whose origin is unclear | | sigaction |
| swapcontext | Removed in POSIX.1-2008 | Portability issues. | Use POSIX threads. |
| tmpnam and tmpnam_r | Marked as obsolete in POSIX.1-2008. | This function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined. | mkstemp, tmpfile |
| ttyslot | Removed in POSIX.1-2001. | | |
| ualarm | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | Errors are under-specified | setitimer or POSIX timer_create |
| usleep | Removed in POSIX.1-2008. | | nanosleep |
| utime | SVr4, POSIX.1-2001. POSIX.1-2008 marks as obsolete. | | |
| valloc | Marked as obsolete in 4.3BSD.<br>Marked as legacy in SUSv2.<br>Removed from POSIX.1-2001 | | posix_memalign |
| vfork | Removed from POSIX.1-2008 | Under-specified in previous standards. | fork |
| wcswcs | This function was not included in the final ISO/IEC 9899:1990/ Amendment 1:1995 (E). | | wcsstr |
| WinExec | WinAPI provides this function only for 16-bit Windows compatibility. | | CreateProcess |
| LoadModule | WinAPI provides this function only for 16-bit Windows compatibility. | | CreateProcess |

#### Fix

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example — Printing Out Time**

```
#include <stdio.h>
#include <time.h>

void timecheck_bad(int argc, char *argv[])
{
    time_t ticks;

    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks));
}
```

In this example, the function `ctime` formats the current time and prints it out. However, `ctime` was removed after C99 because it does not work on multithreaded programs.

**Correction — Different Time Function**

One possible correction is to use `strftime` instead because this function uses a set buffer size.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

void timecheck_good(int argc, char *argv[])
{
    char outBuff[1025];
    time_t ticks;
    struct tm * timeinfo;

    memset(outBuff, 0, sizeof(outBuff));

    ticks = time(NULL);
    timeinfo = localtime(&ticks);
    strftime(outBuff,sizeof(outBuff),"%I:%M%p.",timeinfo);
    fprintf(stdout, outBuff);
}
```

## Check Information

**Category:** API / Function Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-242

# CWE Rule 243

Creation of chroot Jail Without Changing Working Directory

## Description

### Rule Description

*The program uses the chroot() system call to create a jail, but does not change the working directory afterward. This does not prevent access to files outside of the jail.*

### Polyspace Implementation

The rule checker checks for **File manipulation after chroot() without chdir("/")**.

## Examples

### File manipulation after chroot() without chdir("/")

**Issue**

This issue occurs when you have access to a file system outside of the jail created by `chroot`. By calling `chroot`, you create a file system jail that confines access to a specific file subsystem. However, this jail is ineffective if you do not call `chdir("/")`.

**Risk**

If you do not call `chdir("/")` after creating a `chroot` jail, file manipulation functions that takes a path as an argument can access files outside of the jail. An attacker can still manipulate files outside the subsystem that you specified, making the chroot jail ineffective.

**Fix**

After calling `chroot`, call `chdir("/")` to make your `chroot` jail more secure.

**Example — Open File in chroot-jail**

```
#include <unistd.h>
#include <stdio.h>

const char root_path[] = "/var/ftproot";
const char log_path[] = "file.log";
FILE* chrootmisuse() {
    FILE* res;
    chroot(root_path);
    chdir("base");
    res = fopen(log_path, "r");
    return res;
}
```

This example uses `chroot` to create a chroot-jail. However, to use the `chroot` jail securely, you must call `chdir("\")` afterward. This example calls `chdir("base")`, which is not equivalent. Bug Finder also flags `fopen` because `fopen` opens a file in the vulnerable `chroot`-jail.

#### Correction — Call `chdir("/")`

Before opening files, call `chdir("/")`.

```
#include <unistd.h>
#include <stdio.h>

const char root_path[] = "/var/ftproot";
const char log_path[] = "file.log";
FILE* chrootmisuse() {
    FILE* res;
    chroot(root_path);
    chdir("/");
    res = fopen(log_path, "r");
    return res;
}
```

## Check Information

**Category:** Privilege Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-243

# CWE Rule 244

Improper Clearing of Heap Memory Before Release ('Heap Inspection')

## Description

### Rule Description

*Using realloc() to resize buffers that store sensitive information can leave the sensitive information exposed to attack, because it is not removed from memory.*

### Polyspace Implementation

The rule checker checks for **Sensitive heap memory not cleared before release**.

## Examples

### Sensitive heap memory not cleared before release

**Issue**

This issue occurs when dynamically allocated memory contains sensitive data and you do not clear the data before you free the memory.

**Risk**

If the memory zone is reallocated, an attacker can still inspect the sensitive data in the old memory zone.

**Fix**

Before calling `free`, clear out the sensitive data using `memset` or `SecureZeroMemory`.

**Example — Sensitive Buffer Freed, Not Cleared**

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    free(buf);
}
```

In this example, the function uses a buffer of passwords and frees the memory before the end of the function. However, the data in the memory is not cleared by using the `free` command.

**Correction — Nullify Data**

One possible correction is to write over the data to clear out the sensitive information. This example uses `memset` to write over the data with zeros.

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0;i<(sizeof(arr)/sizeof(arr[0]));i++) assert(arr[i]==0)

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);

    if (buf) {
        getpwnam_r(my_user, &pwd, buf, bufsize, &result);
        memset(buf, 0, (size_t)1024);
        isNull(buf);
        free(buf);
    }
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-244

# CWE Rule 248

Uncaught Exception

## Description

### Rule Description

*An exception is thrown from a function, but it is not caught.*

### Polyspace Implementation

The rule checker checks for **Uncaught exception**.

## Examples

### Uncaught exception

#### Issue

This issue occurs when a function that is called in `main()` raises an exception and the exception is not handled. Polyspace highlights the location in the function body where the unhandled exception is raised and flags the call to the function in `main()`. For instance:

```
void foo(){
    throw std::exception(); //Uncaught exception
}
int main(){
    foo(); //Defect
    return 1;
}
```

Polyspace does not raise this defect when an `std::bad_alloc` raised by a `new` operator remains unhandled.

#### Risk

When an exception remains unhandled, the compiler might invoke the function `std::terminate()`, which terminates the program abruptly. The abrupt termination does not invoke any exit handlers, does not call the destructors of the constructed objects, and does not unwind the stack.

Exceptions that are unhandled might result in issues such as memory leaks, security vulnerability, and other unintended behaviors. Poorly designed exception handling process might make your program vulnerable to denial-of-service attacks.

#### Fix

To fix this defect, design the exception handling in your code to handle expected and unexpected exceptions. Call functions that are not `noexcept` in `try` blocks. Handle the exceptions that these functions might raise by using matching `catch()` blocks. Include a `catch-all` block to handle unexpected exceptions.

#### Example — Avoid Unhandled Exceptions

```
#include <exception>
#include <stdexcept>
```

```
int flag();

void foo() {
    if (flag()==0) {
        //....
        throw std::exception();
    }
}

void bar() {
    if (flag()!=0 & flag()!=1) {
        throw std::logic_error("Error");
    }
}



void fubar() {
    foo();
}

int main() {
    foo(); // Defect
    bar(); // Defect
    fubar(); // Defect

}
```

In this exception, the functions `foo()` and `bar()` raise exceptions that are not handled. The function `fubar()` raises an unhandled exception by calling `foo()`. These functions are invoked in `main()`. Because these functions raise exceptions that are unhandled and are called from the `main()` function, Polyspace flags the calls to these functions in `main`.

**Correction — Handle the exceptions**

To resolve this defect, handle the exceptions in your code. For instance, in the `main()` function, call the exception raising function in a `try` block and handle the exception by using a series of `catch()` blocks, including a `catch(...)` block.

```
#include <exception>
#include <stdexcept>

int flag();

void foo() {
    if (flag()==0) {
        //....
        throw std::exception();
    }
}

void bar() {
    if (flag()!=0 & flag()!=1) {
        throw std::logic_error("bla");
    }
}
```

```
void fubar() {
    foo();
}

int main() {
    try{
    foo(); // Defect
    bar(); // Defect
    fubar(); // Defect
    }catch(std::logic_error& e){
        //...
    }catch(std::exception& e){
        //...
    }catch(...){
        //..
    }

}
```

## Check Information

**Category:** Error Conditions, Return Values, Status Codes

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-248

# CWE Rule 252

Unchecked Return Value

## Description

### Rule Description

*The software does not check the return value from a method or function, which can prevent it from detecting unexpected states and conditions.*

### Polyspace Implementation

The rule checker checks for **Returned value of a sensitive function not checked**.

## Examples

### Returned value of a sensitive function not checked

#### Issue

This issue occurs when you call sensitive standard functions that return information about possible errors and you do one of the following:

- Ignore the return value.

  You simply do not assign the return value to a variable, or explicitly cast the return value to `void`.
- Use an output from the function (return value or argument passed by reference) without testing the return value for errors.

The checker considers a function as sensitive if the function call is prone to failure because of reasons such as:

- Exhausted system resources (for example, when allocating resources).
- Changed privileges or permissions.
- Tainted sources when reading, writing, or converting data from external sources.
- Unsupported features despite an existing API.

The checker only considers functions where the *return value* indicates if the function completed without errors.

Some of these functions can perform critical tasks such as:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

#### Risk

If you do not check the return value of functions that perform sensitive tasks and indicate error information through their return values, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

#### Fix

Before continuing with the program, test the return value of critical sensitive functions.

For sensitive functions that are not critical, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

#### Example — Sensitive Function Return Ignored

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);//Noncompliant
}
int read_file(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r");
  return 0; //Noncompliant

}
```

This example shows calls to the sensitive POSIX functions `pthread_attr_init` and `fmemopen`. Their return values are ignored, causing defect.

#### Correction — Cast Function to (void)

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);//Compliant
}
int read_file(int argc, char *argv[])
```

```
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  (void)fmemopen (argv[1], strlen (argv[1]), "r"); //Compliant

  return 0;
}
```

**Correction — Test Return Value**

One possible correction is to test the return value of `pthread_attr_init` and `fmemopen` to check for errors.

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

void initialize() {
    pthread_attr_t attr;

    int result = pthread_attr_init(&attr);//Compliant
    if(result != 0){
        //Handle fatal error
    }
}
int read_file(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r");
  if (in==NULL){
      // Handle error
  }
  return 0;//Compliant
}
```

**Example — Critical Function Return Ignored**

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0));
    pthread_join(thread_id,  &res);
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

**Correction — Test the Return Value of Critical Functions**

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

## Check Information

**Category:** Error Conditions, Return Values, Status Codes

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

CWE-252

# CWE Rule 253

Incorrect Check of Function Return Value

## Description

### Rule Description

*The software incorrectly checks a return value from a function, which prevents the software from detecting errors or exceptional conditions.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Errno not reset**
- **Unprotected dynamic memory allocation**

## Examples

### Errno not reset

#### Issue

This issue occurs when you do not reset errno before calling a function that sets errno to indicate error conditions. However, you check errno for those error conditions after the function call.

#### Risk

The errno is not clean and can contain values from a previous call. Checking errno for errors can give the false impression that an error occurred.

errno is set to zero at program startup but subsequently, errno is not reset by a C standard library function. You must explicitly set errno to zero when required.

#### Fix

Before calling a function that sets errno to indicate error conditions, reset errno to zero explicitly.

#### Example — errno Not Reset Before Call to strtod

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
```

```
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
                  {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

In this example, `errno` is not reset to 0 before the first call to `strtod`. Checking `errno` for 0 later can lead to a false positive.

#### Correction — Reset `errno` Before Call

One possible correction is to reset `errno` to 0 before calling `strtod`.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <float.h>

#define fatal_error() abort()

double func(const char *s1, const char *s2)
{
    double f1;
    errno = 0;
    f1 = strtod (s1, NULL);
    if (0 == errno) {
      double f2 = strtod (s2, NULL);
        if (0 == errno) {
            long double result = (long double)f1 + f2;
            if ((result <= (long double)DBL_MAX) && (result >= (long double)-DBL_MAX))
              {
                return (double)result;
            }
        }
    }
    fatal_error();
    return 0.0;
}
```

### Unprotected dynamic memory allocation

#### Issue

This issue occurs when you access dynamically allocated memory without first checking if the prior memory allocation succeeded.

#### Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

**Fix**

Check the return value of malloc, calloc, or realloc for NULL before accessing the allocated memory location.

```
int *ptr = malloc(size * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

**Example — Unprotected dynamic memory allocation error**

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));

  *p = 2;
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function calloc returns NULL to p. Before accessing the memory through p, the code does not check whether p is NULL

**Correction — Check for NULL Value**

One possible correction is to check whether p has value NULL before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

**Example — Unprotected dynamic memory allocation error only on dereference**

```
#include <stdlib.h>
#include<string.h>
typedef struct recordType {
    const char* id;
    const char* data;
} RECORD;

RECORD* MakerecordType(const char *id,unsigned int size){
    RECORD *rec = (RECORD *)calloc(1, sizeof(RECORD));
    rec->id = strdup(id);

    const char *newData = (char *)calloc(1, size);
    rec->data = newData;
    return rec;
}
```

In this example, the checker raises a defect when you dereference the pointer `rec` without checking for a `NULL` value from the prior dynamic memory allocation.

A similar issue happens with the pointer `newData`. However, a defect is not raised because the pointer is not dereferenced but simply copied over to `rec->data`. Simply copying over a possibly null pointer is not an issue by itself. For instance, callers of the `recordType_new` function might check for `NULL` value of `rec->data` before dereferencing, thereby avoiding a null pointer dereference.

### Check Information

**Category:** Error Conditions, Return Values, Status Codes

## Version History

**Introduced in R2023a**

### See Also

`Check CWE (-cwe)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

CWE-253

# CWE Rule 256

Plaintext storage of a password

## Description

### Rule Definition

*Storing a password in plaintext may result in a system compromise.*

### Polyspace Implementation

This rule checker checks for the issue **Plain Text Password Stored in File System**.

## Examples

### Plain Text Password Stored in File System

#### Issue

**Plain Text Password Stored in File System** occurs when data read from a file is used in functions that expect plain-text passwords. The checker for this issue detects the flow of data from file read functions to the password parameter of functions that take user credentials.

Functions flagged by this checker include the following:

- Windows functions such as `LogonUserW()`, `LogonUserA()` and `CreateProcessWithLogonW()`. The third parameter is the password.
- MySQL functions such as `mysql_real_connect()` and `mysql_real_connect_nonblocking()`. The fourth parameter is the password.

#### Risk

Storing a password in plain-text form in a configuration file is a security risk. Anyone with access to the file can read the passwords and gain access to the password-protected resource.

#### Fix

Instead of reading passwords from a file system, accept passwords on the fly from standard input.

If passwords have to be stored on the file system, store them in encrypted form. After reading an encrypted password from a file, decrypt the password before use in functions that take user credentials. You can use standard encryption and decryption functions from cryptographic libraries, or write your own functions.

#### Extend Checker

You can extend this checker by specifying your own password functions or decryption functions.

Suppose you want to specify the following:

- Function `logOnToServer()` requires an user name and password.

```
void logOnToServer(const char* user,
                 const char*passwd);
```

Suppose the *n_pass*-th argument of this function is the password. For instance, the second argument in the above signature could be the password.

- Function `decrypt()` converts an encrypted password to plain-text form.

  ```
  void decrypt(const char* cipher_text,
               char* plain_text,
               size_t plain_text_size);
  ```

  Suppose the *n_decrypted*-th argument of this function is the decrypted password. For instance, the second argument in the above signature could be the decrypted password.

To make the checker aware of these functions:

1. In a file with extension `.dl`, add the following:

   ```
   .include "models/interfaces/plain_text_password.dl"

   PlainTextPassword.Basic.sensitive("logOnToServer", $InParameterDeref(n_pass-1)).
   PlainTextPassword.Basic.sanitizing("decrypt", $OutParameterDeref(n_decrypted-1)).
   ```

   If *n_pass* and *n_decrypted* are both 2 (that is, the second parameters of each function are the passwords), then the statements become:

   ```
   .include "models/interfaces/plain_text_password.dl"

   PlainTextPassword.Basic.sensitive("logOnToServer", $InParameterDeref(1)).
   PlainTextPassword.Basic.sanitizing("decrypt", $OutParameterDeref(1)).
   ```
2. Specify this file using the option `-code-behavior-specifications`. For instance, if the file is named `passwordFunctions.dl`, use the analysis option:

   ```
   -code-behavior-specifications passwordFunctions.dl
   ```

**Example – Password Read From File**

In this example, the string `plain_passwd` used as a password with `mysql_real_connect()` is previously read from a file `foo.cfg`. If an attacker or malicious user gains access to this file, they can easily access the password-protected resource.

```
#include <mysql/mysql.h>

#define CONNECT_TO_MYSQL_WITH_PASSWORD(passwd) \
        mysql_real_connect( \
        sql, \
        host, \
        user, \
        passwd, \
        db, \
        0, 0x0, 0)

extern MYSQL *sql;
extern const char *host;
extern char *user;
extern char *domain;
extern wchar_t *wuser;
```

```
extern wchar_t *wdomain;
extern char *db;

int non_compliant_from_file_to_mysql() {
    FILE* fp = fopen("foo.cfg", "r");
    if (fp == NULL) {
        return -1;
    }

    char plain_passwd[64];
    if (fgets(plain_passwd, sizeof(plain_passwd), fp) == NULL) {
        fclose(fp);
        return -1;
    }

    CONNECT_TO_MYSQL_WITH_PASSWORD(plain_passwd); //Noncompliant

    memset(plain_passwd, 'X', sizeof(plain_passwd));

    fclose(fp);

    return 0;
}
```

**Correction – Store Encrypted Password in File**

Instead of storing passwords in the clear, you can store passwords in an encrypted form. After reading the encrypted password from a file, you can use a decryption function to decrypt the password on the fly. The example below uses a function `decrypt()` to convert the encrypted password to a plain-text password before using the password with `mysql_real_connect()`.

```
#include <mysql/mysql.h>

#define CONNECT_TO_MYSQL_WITH_PASSWORD(passwd) \
        mysql_real_connect( \
        sql, \
        host, \
        user, \
        passwd, \
        db, \
        0, 0x0, 0)

extern MYSQL *sql;
extern const char *host;
extern char *user;
extern char *domain;
extern wchar_t *wuser;
extern wchar_t *wdomain;
extern char *db;

void decrypt(const char* cipher_text, char* plain_text, size_t plain_text_size);

int non_compliant_from_file_to_mysql() {
    FILE* fp = fopen("foo.cfg", "r");
    if (fp == NULL) {
        return -1;
    }
    char plain_passwd[64], cipher_passwd[64];
    if (fgets(cipher_passwd, sizeof(cipher_passwd), fp) == NULL) {
        fclose(fp);
        return -1;
    }

    decrypt(cipher_passwd, plain_passwd, sizeof(plain_passwd));

    CONNECT_TO_MYSQL_WITH_PASSWORD(plain_passwd);

    memset(plain_passwd, 'X', sizeof(plain_passwd));

    fclose(fp);

    return 0;
}
```

To make the Polyspace analysis aware of the nature of your decryption function:

1. Specify the following in a file with extension `.dl` (for instance, `decryptFunctions.dl`):

   ```
   .include "models/interfaces/plain_text_password.dl"

   PlainTextPassword.Basic.sanitizing("decrypt", $OutParameterDeref(1))..
   ```
2. Specify the following option with the analysis:

   ```
   -code-behavior-specifications decryptFunctions.dl
   ```

   See also `-code-behavior-specifications`.

## Check Information

**Category:** Credentials Management Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-256

# CWE Rule 312

Cleartext Storage of Sensitive Information

## Description

### Rule Description

*The product stores sensitive information in cleartext within a resource that might be accessible to another control sphere.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Sensitive heap memory not cleared before release**
- **Uncleared sensitive data in stack**

## Examples

### Sensitive heap memory not cleared before release

#### Issue

This issue occurs when dynamically allocated memory contains sensitive data and you do not clear the data before you free the memory.

#### Risk

If the memory zone is reallocated, an attacker can still inspect the sensitive data in the old memory zone.

#### Fix

Before calling `free`, clear out the sensitive data using `memset` or `SecureZeroMemory`.

#### Example — Sensitive Buffer Freed, Not Cleared

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    free(buf);
}
```

In this example, the function uses a buffer of passwords and frees the memory before the end of the function. However, the data in the memory is not cleared by using the `free` command.

#### Correction — Nullify Data

One possible correction is to write over the data to clear out the sensitive information. This example uses `memset` to write over the data with zeros.

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0;i<(sizeof(arr)/sizeof(arr[0]));i++) assert(arr[i]==0)

void sensitiveheapnotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char* buf = (char*) malloc(1024);

    if (buf) {
        getpwnam_r(my_user, &pwd, buf, bufsize, &result);
        memset(buf, 0, (size_t)1024);
        isNull(buf);
        free(buf);
    }
}
```

### Uncleared sensitive data in stack

#### Issue

This issue occurs when statically allocated memory contains sensitive data and you do not clear the data before exiting a function or program.

#### Risk

Leaving sensitive information in your stack, such as passwords or user information, allows an attacker additional access to the information after your program has ended.

#### Fix

Before exiting a function or program, clear out the memory zones that contain sensitive data by using `memset` or `SecureZeroMemory`.

#### Example — Static Buffer of Password Information

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>

void bug_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
}
```

In this example, a static buffer is filled with password information. The program frees the stack memory at the end of the program. However, the data is still accessible from the memory.

#### Correction — Clear Memory

One possible correction is to write over the memory before exiting the function. This example uses `memset` to clear the data from the buffer memory.

```
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <pwd.h>
#include <assert.h>

#define isNull(arr) for(int i=0;i<(sizeof(arr)/sizeof(arr[0]));i++) assert(arr[i]==0)

void corrected_sensitivestacknotcleared(const char * my_user) {
    struct passwd* result, pwd;
    long bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    char buf[1024] = "";
    getpwnam_r(my_user, &pwd, buf, bufsize, &result);
    memset(buf, 0, (size_t)1024);
    isNull(buf);
}
```

## Check Information

**Category:** Information Management Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-312

# CWE Rule 335

Incorrect Usage of Seeds in Pseudo-Random Number Generator (PRNG)

## Description

### Rule Description

*The software uses a Pseudo-Random Number Generator (PRNG) but does not correctly manage seeds.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Deterministic random output from constant seed**
- **Hard-coded sensitive data**
- **Predictable random output from predictable seed**

## Examples

### Deterministic random output from constant seed

**Issue**

This issue occurs when you use standard random number generator functions that have deterministic output given a constant seed.

The checker detects this issue with the following random number generator functions:

- C Standard Library functions such as `srand`, `srandom` and `initstate`
- OpenSSL functions such as `RAND_seed` and `RAND_add`
- C++ Standard Library functions such as `std::linear_congruential_engine<>::seed()` and `std::mersenne_twister_engine<>::seed()` (and also the constructors of these class templates)

**Risk**

With constant seeds, random number generator functions produce the same output every time your program is run. A hacker can disrupt your program if they know how your program behaves.

**Fix**

Use a different random standard function or use a nonconstant seed.

Some standard random routines are inherently cryptographically weak on page 13-97, and should not be used for security purposes.

**Example — Random Number Generator Initialization**

```
#include <stdlib.h>

void random_num(void)
```

```
{
    srand(12345U);
    /* ... */
}
```

This example initializes a random number generator using `srand` with a constant seed. The random number generation is deterministic, making this function cryptographically weak.

#### Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <stdio.h>

unsigned int random_num_time(void)
{

    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

### Hard-coded sensitive data

#### Issue

This issue occurs when data that is potentially sensitive is directly exposed in the code, for instance, as string literals. The checker identifies certain data as sensitive from their use in certain functions such as password encryption functions.

Following data can be potentially sensitive.

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Host name | • sethostname, setdomainname, gethostbyname, gethostbyname2, getaddrinfo, gethostbyname_r, gethostbyname2_r (string argument)<br>• inet_aton, inet_pton, inet_net_pton, inet_addr, inet_network (string argument)<br>• mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (2nd argument) |
| Password | • CreateProcessWithLogonW, LogonUser (1st argument)<br>• mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (3rd argument) |
| Database | • MySQL: mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (4th argument)<br>• SQLite: sqlite3_open, sqlite3_open16, sqlite3_open_v2 (1st argument)<br>• PostgreSQL: PQconnectdb<br>• Microsoft SQL: SQLDriverConnect (3rd argument) |
| User name | • getpw, getpwnam, getpwnam_r, getpwuid, getpwuid_r |
| Salt | crypt, crypt_r (2nd argument) |
| Cryptography keys and initialization vectors | OpenSSL:<br>• EVP_CipherInit, EVP_EncryptInit, EVP_DecryptInit (3rd argument)<br>• EVP_CipherInit_ex, EVP_EncryptInit_ex, EVP_DecryptInit_ex (4th argument) |
| Seed | • srand, srandom, initstate (1st argument)<br>• OpenSSL: RAND_seed, RAND_add |

**Risk**

Information that is hardcoded can be queried from binaries generated from the code.

**Fix**

Avoid hard coding sensitive information.

**Example — Sensitive Data Exposed Through String Literals**

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"
const char *host = "localhost";
char *user = "guest";
char *passwd;

DbHandle connect_to_database_server(const char *db)
{
    passwd = (char*)"guest";
    return (DbHandle)
        mysql_real_connect (sql, host, user, passwd, db, 0, 0x0, 0);
}
```

In this example, the arguments `host` (host name), `user` (user name), and `passwd` (password) are string literals and directly exposed in the code.

Querying the generated binary for ASCII strings can reveal this information.

**Correction – Read Sensitive Data from Secured Configuration Files**

One possible correction is to read the data from a configuration file. In the following corrected example, the call to function `connect_to_database_server_init` presumably reads the host name, user name, and password into its arguments from a secured configuration file.

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"

int connect_to_database_server_init(const char **host,
                                    const char **user,
                                    const char **passwd,
                                    const char **db);
```

```
DbHandle connect_to_database_server(const char *db)
{
    const char *host_from_cfg;
    const char *user_from_cfg;
    const char *passwd_from_cfg;
    const char *db_from_cfg;
    if (connect_to_database_server_init(&host_from_cfg,
                                        &user_from_cfg,
                                        &passwd_from_cfg,
                                        &db_from_cfg))
    {
        return (DbHandle)
            mysql_real_connect (sql, host_from_cfg, user_from_cfg,
                        passwd_from_cfg, db_from_cfg,  0, 0x0, 0);
    }
    else
        return (DbHandle)0x0;
}
```

### Predictable random output from predictable seed

**Issue**

This issue occurs when you use standard random number generator functions with a nonconstant but predictable seed. Examples of predictable seed generators are `time`, `gettimeofday`, and `getpid`.

The checker detects this issue with the following random number generator functions:

- C Standard Library functions such as `srand`, `srandom` and `initstate`
- C++ Standard Library functions such as `std::linear_congruential_engine<>::seed()` and `std::mersenne_twister_engine<>::seed()` (and also the constructors of these class templates)

**Risk**

When you use predictable seed values for random number generation, your random numbers are also predictable. A hacker can disrupt your program if they know how your program behaves.

**Fix**

You can use a different function to generate less predictable seeds.

You can also use a different random number generator that does not require a seed. For example, the Windows API function `rand_s` seeds itself by default. It uses information from the entire system, for example, system time, thread ids, system counter, and memory clusters. This information is more random and a user cannot access this information.

Some standard random routines are inherently cryptographically weak on page 13-97, and should not be used for security purposes.

**Example — Seed as an Argument**

```
#include <stdlib.h>
#include <time.h>

void seed_rng(int seed)
{
    srand(seed);
}

int generate_num(void)
{
    seed_rng(time(NULL) + 3);
    /* ... */
}
```

This example uses `srand` to start the random number generator with `seed` as the seed. However, `seed` is predictable because the function `time` generates it. So, an attacker can predict the random numbers generated by `srand`.

#### Correction — Use Different Random Number Generator

One possible correction is to use a random number generator that does not require a seed. This example uses `rand_s`.

```
#define _CRT_RAND_S

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int generate_num(void)
{
    unsigned int number;
    errno_t err;
    err = rand_s(&number);

    if(err != 0)
    {
        return number;
    }
    else
    {
        return err;
    }
}
```

## Check Information

**Category:** Random Number Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**

CWE-335

# CWE Rule 338

Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)

## Description

### Rule Description

*The product uses a Pseudo-Random Number Generator (PRNG) in a security context, but the PRNG's algorithm is not cryptographically strong.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Predictable block cipher initialization vector**
- **Predictable cipher key**
- **Vulnerable pseudo-random number generator**

## Examples

### Predictable block cipher initialization vector

#### Issue

This issue occurs when you use a weak random number generator for the block cipher initialization vector.

#### Risk

If you use a weak random number generator for the initiation vector, your data is vulnerable to dictionary attacks.

Block ciphers break your data into blocks of fixed size. Block cipher modes such as CBC (Cipher Block Chaining) protect against dictionary attacks by XOR-ing each block with the encrypted output from the previous block. To protect the first block, these modes use a random initialization vector (IV). If you use a weak random number generator for your IV, your data becomes vulnerable to dictionary attacks.

#### Fix

Use a strong pseudo-random number generator (PRNG) for the initialization vector. For instance, use:

- OS-level PRNG such as `/dev/random` on UNIX or `CryptGenRandom()` on Windows
- Application-level PRNG such as Advanced Encryption Standard (AES) in Counter (CTR) mode, HMAC-SHA1, etc.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

#### Example — Predictable Initialization Vector

```
#include <openssl/evp.h>
#include <openssl/rand.h>
```

```
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_pseudo_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the function `RAND_pseudo_bytes` declared in `openssl/rand.h` produces the initialization vector. The byte sequences that `RAND_pseudo_bytes` generates are not necessarily unpredictable.

**Correction — Use Strong Random Number Generator**

Use a strong random number generator to produce the initialization vector. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *key){
    unsigned char iv[SIZE16];
    RAND_bytes(iv, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

## Predictable cipher key

### Issue

This issue occurs when you use a weak random number generator for the encryption or decryption key.

### Risk

If you use a weak random number generator for the encryption or decryption key, an attacker can retrieve your key easily.

You use a key to encrypt and later decrypt your data. If a key is easily retrieved, data encrypted using that key is not secure.

### Fix

Use a strong pseudo-random number generator (PRNG) for the key. For instance:

- Use an OS-level PRNG such as `/dev/random` on UNIX or `CryptGenRandom()` on Windows
- Use an application-level PRNG such as Advanced Encryption Standard (AES) in Counter (CTR) mode, HMAC-SHA1, etc.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

#### Example — Predictable Cipher Key

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_pseudo_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the function `RAND_pseudo_bytes` declared in `openssl/rand.h` produces the cipher key. However, the byte sequences that `RAND_pseudo_bytes` generates are not necessarily unpredictable.

#### Correction — Use Strong Random Number Generator

One possible correction is to use a strong random number generator to produce the cipher key. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

### Vulnerable pseudo-random number generator

#### Issue

This issue occurs when you use cryptographically weak pseudo-random number generator (PRNG) routines.

The list of cryptographically weak routines flagged by this checker include:

- `rand`, `random`
- `drand48`, `lrand48`, `mrand48`, `erand48`, `nrand48`, `jrand48`, and their `_r` equivalents such as `drand48_r`
- `RAND_pseudo_bytes`

#### Risk

These cryptographically weak routines are predictable and must not be used for security purposes. When a predictable random value controls the execution flow, your program is vulnerable to malicious attacks.

**Fix**

Use more cryptographically sound random number generators, such as `CryptGenRandom` (Windows), `OpenSSL/RAND_bytes`(Linux/UNIX).

**Example — Random Loop Numbers**

```
#include <stdio.h>
#include <stdlib.h>

volatile int rd = 1;
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;

    nloops = rand();

    for (j = 0; j < nloops; j++) {
        if (random_r(&buf, &i))
            exit(1);
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

This example uses `rand` and `random_r` to generate random numbers. If you use these functions for security purposes, these PRNGs can be the source of malicious attacks.

**Correction — Use Stronger PRNG**

One possible correction is to replace the vulnerable PRNG with a stronger random number generator.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rand.h>

volatile int rd = 1;
int main(int argc, char* argv[])
{
    int j, r, nloops;
    unsigned char buf;
    unsigned int seed;
    int i = 0;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <seed> <nloops>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    seed = atoi(argv[1]);
    nloops = atoi(argv[2]);

    for (j = 0; j < nloops; j++) {
        if (RAND_bytes(&buf, i) != 1)
```

```
            exit(1);
        printf("RAND_bytes: %u\n", (unsigned)buf);
    }
    return 0;
}
```

## Check Information

**Category:** Random Number Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-338

# CWE Rule 353

Missing Support for Integrity Check

## Description

### Rule Description

*The software uses a transmission protocol that does not include a mechanism for verifying the integrity of the data during transmission, such as a checksum.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Context initialized incorrectly for digest operation**
- **Nonsecure hash algorithm**

## Examples

### Context initialized incorrectly for digest operation

#### Issue

This issue occurs when you initialize an `EVP_MD_CTX` context object for a specific digest operation but use the context for a different operation.

For instance, you initialize the context for creating a message digest only.

```
ret = EVP_DigestInit(ctx, EVP_sha256())
```

However, you perform a final step for signing:

```
ret = EVP_SignFinal(&ctx, out, &out_len, pkey);
```

The error is shown only if the final step is not consistent with the initialization of the context. If the intermediate update steps are inconsistent, it does not trigger an error because the intermediate steps do not depend on the nature of the operation. For instance, `EVP_DigestUpdate` works identically to `EVP_SignUpdate`.

#### Risk

Mixing up different operations on the same context can lead to obscure code. It is difficult to determine at a glance whether the current object is used for message digest creation, signing, or verification. The mixup can also lead to a failure in the operation or unexpected message digest.

#### Fix

After you set up a context for a certain family of operations, use the context for only that family of operations. For instance, use these pairs of functions for initialization and final steps.

- `EVP_DigestInit` : `EVP_DigestFinal`
- `EVP_DigestInit_ex` : `EVP_DigestFinal_ex`

- EVP_DigestSignInit : EVP_DigestSignFinal

If you want to reuse an existing context object for a different family of operations, reinitialize the context.

**Example — Inconsistent Initial and Final Digest Operation**

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf16;
unsigned int out_len16;

void func(unsigned char *src, size_t len){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_sha256(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_SignUpdate(ctx, src, len);
  if (ret != 1) fatal_error();

  ret = EVP_DigestSignFinal(ctx, out_buf16, (size_t*) out_len16);

  if (ret != 1) fatal_error();
}
```

In this example, the context object is initialized for signing only with EVP_SignInit but the final step attempts to create a signed digest with EVP_DigestSignFinal.

**Correction — Use One Family of Operations**

One possible correction is to use the context object for signing only. Change the final step to EVP_SignFinal in keeping with the initialization step.

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf16;
unsigned int out_len16;

void corrected_cryptomdbadfunction(unsigned char *src, size_t len, EVP_PKEY* pkey){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_sha256(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_SignUpdate(ctx, src, len);
  if (ret != 1) fatal_error();

  ret = EVP_SignFinal(ctx, out_buf16, &out_len16, pkey);
  if (ret != 1) fatal_error();
}
```

### Nonsecure hash algorithm

**Issue**

This issue occurs when you use a cryptographic hash function that is proven to be weak against certain forms of attack.

The hash functions flagged by this checker include SHA-0, SHA-1, MD4, MD5, and RIPEMD-160. The checker detects the use of these hash functions in:

- Functions from the EVP API such as `EVP_DigestUpdate` or `EVP_SignUpdate`.
- Functions from the low level API such as `SHA1_Update` or `MD5_Update`.

**Risk**

You use a hash function to create a message digest from input data and thereby ensure integrity of your data. The hash functions flagged by this checker use algorithms with known weaknesses that an attacker can exploit. The attacks can comprise the integrity of your data.

**Fix**

Use a more secure hash function. For instance, use the later SHA functions such as SHA-224, SHA-256, SHA-384, and SHA-512.

**Example — Use of MD5 Algorithm**

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
unsigned int out_len;

void func(unsigned char *src, size_t len, EVP_PKEY* pkey){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_md5(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_DigestUpdate(ctx,src,len);

  if (ret != 1) fatal_error();

  ret = EVP_SignFinal(ctx, out_buf, &out_len, pkey);
  if (ret != 1) fatal_error();
}
```

In this example, during initialization with `EVP_SignInit_ex`, the context object is associated with the weak hash function MD5. The checker flags the usage of this context in the update step with `EVP_DigestUpdate`.

**Correction — Use SHA-2 Family Function**

One possible correction is to use a hash function from the SHA-2 family, such as SHA-256.

```
#include <openssl/evp.h>
```

```
#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
unsigned int out_len;

void func(unsigned char *src, size_t len, EVP_PKEY* pkey){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_sha256(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_SignUpdate(ctx, src, len);
  if (ret != 1) fatal_error();

  ret = EVP_SignFinal(ctx, out_buf, &out_len, pkey);
  if (ret != 1) fatal_error();
}
```

## Check Information

**Category:** Data Integrity Issues

# Version History

**Introduced in R2023a**

## See Also

Check CWE (-cwe)

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-353

# CWE Rule 362

Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

## Description

### Rule Description

*The program contains a code sequence that can run concurrently with other code, and the code sequence requires temporary, exclusive access to a shared resource, but a timing window exists in which the shared resource can be modified by another code sequence that is operating concurrently.*

### Polyspace Implementation

The rule checker checks for **File descriptor exposure to child process**.

## Examples

### File descriptor exposure to child process

**Issue**

This issue occurs when a process is forked and the child process uses file descriptors inherited from the parent process.

**Risk**

When you fork a child process, file descriptors are copied from the parent process, which means that you can have concurrent operations on the same file. Use of the same file descriptor in the parent and child processes can lead to race conditions that may not be caught during standard debugging. If you do not properly manage the file descriptor permissions and privileges, the file content is vulnerable to attacks targeting the child process.

**Fix**

Check that the file has not been modified before forking the process. Close all inherited file descriptors and reopen them with stricter permissions and privileges, such as read-only permission.

**Example — File Descriptor Accessed from Forked Process**

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <fcntl.h>
# include <sys/types.h>
# include <sys/stat.h>



const char *test_file="/home/user/test.txt";

void func(void)
{
    char c;
    pid_t pid;
    /* create file descriptor in read and write mode */
    int fd = open(test_file, O_RDWR);
    if (fd == -1)
    {
        /* Handle error */
        abort();
    }
    /* fork process */
    pid = fork();
    if (pid == -1)
    {
        /* Handle error */
        abort();
    }
    else if (pid == 0)
    {   /* Child process accesses file descriptor inherited
        from parent process */
        (void)read(fd, &c, 1);
    }
    else
    {   /* Parent process access same file descriptor as
        child process */
        (void)read(fd, &c, 1);
    }
}
```

In this example, a file descriptor `fd` is created in read and write mode. The process is then forked. The child process inherits and accesses `fd` with the same permissions as the parent process. A race condition exists between the parent and child processes. The contents of the file is vulnerable to attacks through the child process.

#### Correction — Close and Reopen Inherited File Descriptor

After you create the file descriptor, check the file for tampering. Then, close the inherited file descriptor in the child process and reopen it in read-only mode.

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <fcntl.h>
# include <sys/types.h>
# include <sys/stat.h>


const char *test_file="/home/user/test.txt";

void func(void)
{
    char c;
    pid_t pid;

    /* Get the state of file for further file tampering checking */

    /* create file descriptor in read and write mode */
    int fd = open(test_file, O_RDWR);
    if (fd == -1)
    {
        /* Handle error */
        abort();
    }

    /* Be sure the file was not tampered with while opening */

    /* fork process */

    pid = fork();
    if (pid == -1)
    {
        /* Handle error */
        (void)close(fd);
        abort();
    }
    else if (pid == 0)
    {  /* Close file descriptor in child process and repoen
        it in read only mode */

        (void)close(fd);
        fd = open(test_file, O_RDONLY);
        if (fd == -1)
        {
            /* Handle error */
            abort();
        }


        (void)read(fd, &c, 1);
        (void)close(fd);
    }
    else
    {  /* Parent acceses original file descriptor */
        (void)read(fd, &c, 1);
        (void)close(fd);
    }
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

Check CWE (-cwe)

### Topics

"Check for and Review Coding Standard Violations"

**External Websites**
CWE-362

# CWE Rule 364

Signal Handler Race Condition

## Description

### Rule Description

*The software uses a signal handler that introduces a race condition.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Function called from signal handler not asynchronous-safe**
- **Function called from signal handler not asynchronous-safe (strict ISO C)**
- **Shared data access within signal handler**

## Examples

### Function called from signal handler not asynchronous-safe

#### Issue

This issue occurs when a signal handler calls a function that is not asynchronous-safe according to the POSIX standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

#### Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

#### Fix

The POSIX standard defines these functions as asynchronous-safe. You can call these functions from a signal handler.

| | | |
|---|---|---|
| `_exit()` | `getpgrp()` | `setsockopt()` |
| `_Exit()` | `getpid()` | `setuid()` |
| `abort()` | `getppid()` | `shutdown()` |
| `accept()` | `getsockname()` | `sigaction()` |
| `access()` | `getsockopt()` | `sigaddset()` |

| | | |
|---|---|---|
| aio_error() | getuid() | sigdelset() |
| aio_return() | kill() | sigemptyset() |
| aio_suspend() | link() | sigfillset() |
| alarm() | linkat() | sigismember() |
| bind() | listen() | signal() |
| cfgetispeed() | lseek() | sigpause() |
| cfgetospeed() | lstat() | sigpending() |
| cfsetispeed() | mkdir() | sigprocmask() |
| cfsetospeed() | mkdirat() | sigqueue() |
| chdir() | mkfifo() | sigset() |
| chmod() | mkfifoat() | sigsuspend() |
| chown() | mknod() | sleep() |
| clock_gettime() | mknodat() | sockatmark() |
| close() | open() | socket() |
| connect() | openat() | socketpair() |
| creat() | pathconf() | stat() |
| dup() | pause() | symlink() |
| dup2() | pipe() | symlinkat() |
| execl() | poll() | sysconf() |
| execle() | posix_trace_event() | tcdrain() |
| execv() | pselect() | tcflow() |
| execve() | pthread_kill() | tcflush() |
| faccessat() | pthread_self() | tcgetattr() |
| fchdir() | pthread_sigmask() | tcgetpgrp() |
| fchmod() | quick_exit() | tcsendbreak() |
| fchmodat() | raise() | tcsetattr() |
| fchown() | read() | tcsetpgrp() |
| fchownat() | readlink() | time() |
| fcntl() | readlinkat() | timer_getoverrun() |
| fdatasync() | recv() | timer_gettime() |
| fexecve() | recvfrom() | timer_settime() |
| fork() | recvmsg() | times() |
| fpathconf() | rename() | umask() |
| fstat() | renameat() | uname() |
| fstatat() | rmdir() | unlink() |
| fsync() | select() | unlinkat() |
| ftruncate() | sem_post() | utime() |

| | | |
|---|---|---|
| futimens() | send() | utimensat() |
| getegid() | sendmsg() | utimes() |
| geteuid() | sendto() | wait() |
| getgid() | setgid() | waitpid() |
| getgroups() | setpgid() | write() |
| getpeername() | setsid() | |

Functions not in the previous table are not asynchronous-safe, and should not be called from a signal hander.

**Example — Call to printf() Inside Signal Handler**

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler(int signum)
{
    /* Call function printf() that is not
    asynchronous-safe */
    printf("signal %d received.", signum);
    e_flag = 1;
}

int main(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, sizeof(char));
    if (info == NULL)
    {
        /* Handle Error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

In this example, `sig_handler` calls `printf()` when catching a signal. If the handler catches another signal while `printf()` is executing, the behavior of the program is undefined.

**Correction — Set Flag Only in Signal Handler**

Use your signal handler to set only the value of a flag. `e_flag` is of type volatile `sig_atomic_t`. `sig_handler` can safely access it asynchronously.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler1(int signum)
{
    int s0 = signum;
    e_flag = 1;
}

int func(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler1) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, 1);
    if (info == NULL)
    {
        /* Handle error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

### Function called from signal handler not asynchronous-safe (strict ISO C)

#### Issue

This issue occurs when a signal handler calls a function that is not asynchronous-safe according to the C standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

The C standard defines a stricter subset of functions as asynchronous-safe compared to the set of functions that are asynchronous-safe according to the POSIX standard. **Function called from signal handler not asynchronous-safe (strict ISO C)** reports a defect when a signal handler calls any function that is not part of that subset, even if the function is asynchronous-safe according to the POSIX standard.

To check for calls to functions that are not asynchronous-safe according to the POSIX standard, enable checker **Function called from signal handler not asynchronous-safe**.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

#### Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

#### Fix

The C standard defines the following functions as asynchronous-safe. You can call these functions from a signal handler:

- `abort()`
- `_Exit()`
- `quick_exit()`
- `signal()`

#### Example — Call to raise() Inside Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
```

```
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}

void sig_handler(int signum)
{
    int s0 = signum;
    /* Call raise() */
    if (raise(SIGTERM) != 0) {
        /* Handle error */
    }
}

int finc(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` calls `raise()` when catching a signal. If the handler catches another signal while `raise()` is executing, the behavior of the program is undefined.

**Correction — Remove Call to `raise()` in Signal Handler**

According to the C standard, the only functions that you can safely call from a signal handler are `abort()`, `_Exit()`, `quick_exit()`, and `signal()`.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}
void sig_handler(int signum)
{
    int s0 = signum;


}

int func(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

## Shared data access within signal handler

### Issue

This issue occurs when you access or modify a shared object inside a signal handler.

### Risk

When you define a signal handler function to access or modify a shared object, the handler accesses or modifies the shared object when it receives a signal. If another function is already accessing the shared object, that function causes a race condition and can leave the data in an inconsistent state.

### Fix

To access or modify shared objects inside a signal handler, check that the objects are lock-free atomic, or, if they are integers, declare them as `volatile sig_atomic_t`.

### Example — int Variable Access in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* declare global variable. */
int e_flag;

void sig_handler(int signum)
{
    /* Signal handler accesses variable that is not
    of type volatile sig_atomic_t. */
    e_flag = signum;
}
```

```
int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` accesses `e_flag`, a variable of type `int`. A concurrent access by another function can leave `e_flag` in an inconsistent state.

**Correction — Declare Variable of Type `volatile sig_atomic_t`**

Before you access a shared variable from a signal handler, declare the variable with type `volatile sig_atomic_t` instead of `int`. You can safely access variables of this type asynchronously.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* Declare variable of type volatile sig_atomic_t. */
volatile sig_atomic_t e_flag;
void sig_handler(int signum)
{
    /* Use variable of proper type inside signal handler. */
    e_flag = signum;

}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

## Check Information

**Category:** Signal Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-364

# CWE Rule 366

Race Condition within a Thread

## Description

### Rule Description

*If two threads of execution use a resource simultaneously, there exists the possibility that resources may be used while invalid, in turn making the state of execution undefined.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Atomic load and store sequence not atomic**
- **Atomic variable accessed twice in an expression**
- **Data race**
- **Data race on adjacent bit fields**
- **Data race through standard library function call**

## Examples

### Atomic load and store sequence not atomic

#### Issue

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This issue occurs when you use these functions to load, and then store an atomic variable.

- C functions:
  - `atomic_load()`
  - `atomic_load_explicit()`
  - `atomic_store()`
  - `atomic_store_explicit()`
- C++ functions:
  - `std::atomic_load()`
  - `std::atomic_load_explicit()`
  - `std::atomic_store()`
  - `std::atomic_store_explicit()`
  - `std::atomic::load()`
  - `std::atomic::store()`

A thread cannot interrupt an atomic load or an atomic store operation on a variable, but a thread can interrupt a store, and then load sequence.

**Risk**

A thread can modify a variable between the load and store operations, resulting in a data race condition.

**Fix**

To read, modify, and store a variable atomically, use a compound assignment operator such as +=, `atomic_compare_exchange()` or `atomic_fetch_*`-family functions.

**Example — Loading Then Storing an Atomic Variable**

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void init_flag(void)
{
    atomic_init(&flag, false);
}

void toggle_flag(void)
{
    bool temp_flag = atomic_load(&flag);
    temp_flag = !temp_flag;
    atomic_store(&flag, temp_flag);
}

bool get_flag(void)
{
    return atomic_load(&flag);
}
```

In this example, variable `flag` of type `atomic_bool` is referenced twice inside the `toggle_flag()` function. The function loads the variable, negates its value, then stores the new value back to the variable. If two threads call `toggle_flag()`, the second thread can access `flag` between the load and store operations of the first thread. `flag` can end up in an incorrect state.

**Correction — Use Compound Assignment to Modify Variable**

One possible correction is to use a compound assignment operator to toggle the value of `flag`. The C standard defines the operation by using ^= as atomic.

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void toggle_flag(void)
{
    flag ^= 1;
}

bool get_flag(void)
{
    return flag;
}
```

### Atomic variable accessed twice in an expression

#### Issue

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

This issue occurs when C atomic types or C++ `std::atomic` class variables appear twice in an expression and there are:

- Two atomic read operations on the variable.
- An atomic read and a distinct atomic write operation on the variable.

The C standard defines certain operations on atomic variables that are thread safe and do not cause data race conditions. Unlike individual operations, a pair of operations on the same atomic variable in an expression is not thread safe.

#### Risk

A thread can modify the atomic variable between the pair of atomic operations, which can result in a data race condition.

#### Fix

Do not reference an atomic variable twice in the same expression.

#### Example — Referencing Atomic Variable Twice in an Expression

```
#include <stdatomic.h>

atomic_int n = ATOMIC_VAR_INIT(0);

int compute_sum(void)
{
    return n * (n + 1) / 2;
}
```

In this example, the global variable `n` is referenced twice in the return statement of `compute_sum()`. The value of `n` can change between the two distinct read operations. `compute_sum()` can return an incorrect value.

#### Correction — Pass Variable as Function Argument

One possible correction is to pass the variable as a function argument `n`. The variable is copied to memory and the read operations on the copy guarantee that `compute_sum()` returns a correct result. If you pass a variable of type `int` instead of type `atomic_int`, the correction is still valid.

```
#include <stdatomic.h>

int compute_sum(atomic_int n)
{
    return n * (n + 1) / 2;
}
```

### Data race

#### Issue

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

This issue occurs when:

1. Multiple tasks perform unprotected operations on a shared variable.
2. At least one task performs a write operation.
3. At least one operation is nonatomic. To detect data race on both atomic and nonatomic operations, use the options `-detect-atomic-data-race`. See “Extend Data Race Checkers to Atomic Operations”.

   See “Define Atomic Operations in Multitasking Code”.

If you activate this checker without specifying the multitasking options first, you see a warning in the log:To find this defect, specify the multitasking options before analysis.

To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

#### Risk

Data race can result in unpredictable values of the shared variable because you do not control the order of the operations in different tasks.

Data races between two write operations are more serious than data races between a write and read operation. Two write operations can interfere with each other and result in indeterminate values. To identify write-write conflicts, use the filters on the **Detail** column of the **Results List** pane. For these conflicts, the **Detail** column shows the additional line:

```
 Variable value may be altered by write-write concurrent access.
```

See also “Filter and Group Results in Polyspace Desktop User Interface” or “Filter and Sort Results in Polyspace Access Web Interface”.

#### Fix

To fix this defect, protect the operations on the shared variable using critical sections, temporal exclusion or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon. For an example, see below.

#### Extend Checker

Extend this checker to check for data races in operations that Bug Finder might not detect by default. For instance:

- You might be using multithreading functions that are not supported by Polyspace. Extend this checker by mapping the functions of your multithreading functions to their known POSIX equivalent. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.
- Polyspace assumes that certain operations are atomic and excludes them from data race checks. See “Define Atomic Operations in Multitasking Code”. These assumptions might not apply to your environment. To extend the data race checkers to include these operations, use the option `-detect-atomic-data-race`. See “Extend Data Race Checkers to Atomic Operations”.

#### Example — Unprotected Operation on Global Variable from Multiple Tasks

```
int var;
void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
    var++;
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Specification | |
|---|---|---|
| Configure multitasking manually | ☑ | |
| Tasks (-entry-points) | task1<br>task2<br>task3 | |
| Critical section details (-critical-section-begin -critical-section-end) | **Starting routine** | **Ending routine** |
| | begin_critical_section | end_critical_section |

On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2,task3
  -critical-section-begin begin_critical_section:cs1
  -critical-section-end end_critical_section:cs1
```

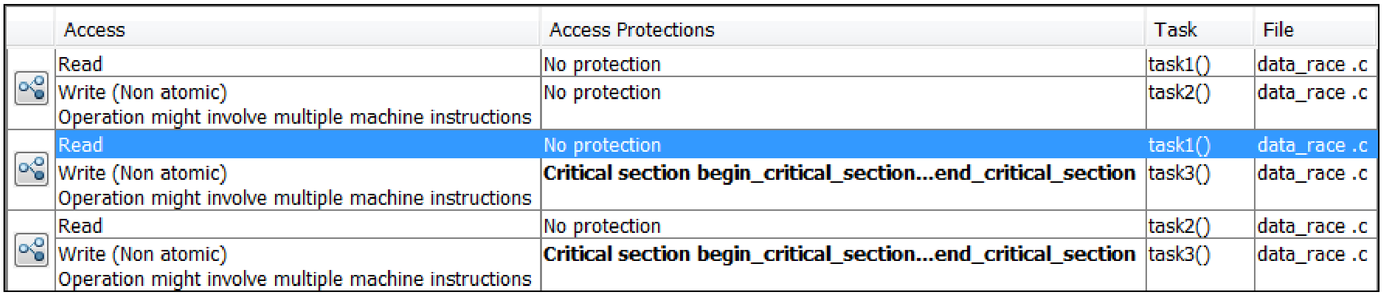
In this example, the tasks `task1`, `task2`, and `task3` call the function `increment`. `increment` contains the operation `var++` that can involve multiple machine instructions including:

- Reading `var`.
- Writing an increased value to `var`.

These machine instructions, when executed from `task1` and `task2`, can occur concurrently in an unpredictable sequence. For example, reading `var` from `task1` can occur either before or after writing to `var` from `task2`. Therefore the value of `var` can be unpredictable.

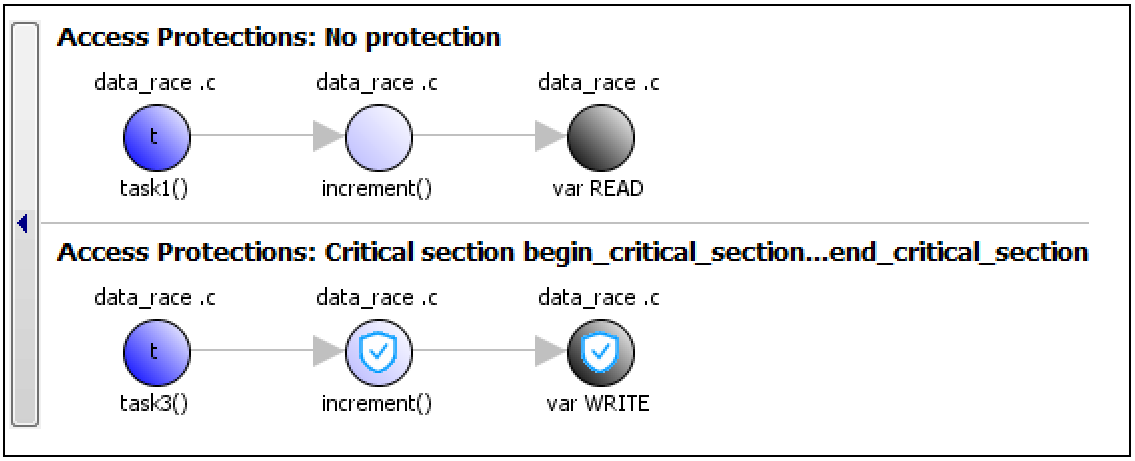
Though `task3` calls `increment` inside a critical section, other tasks do not use the same critical section. The operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

Therefore, the three tasks are operating on a shared variable without common protection. In your result details, you see each pair of conflicting function calls.



| | Access | Access Protections | Task | File |
|---|---|---|---|---|
| | Read | No protection | task1() | data_race .c |
| | Write (Non atomic)<br>Operation might involve multiple machine instructions | No protection | task2() | data_race .c |
| | Read | No protection | task1() | data_race .c |
| | Write (Non atomic)<br>Operation might involve multiple machine instructions | **Critical section begin_critical_section...end_critical_section** | task3() | data_race .c |
| | Read | No protection | task2() | data_race .c |
| | Write (Non atomic)<br>Operation might involve multiple machine instructions | **Critical section begin_critical_section...end_critical_section** | task3() | data_race .c |

If you click the icon, you see the function call sequence starting from the entry point to the read or write operation. You also see that the operation starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.



#### Correction — Place Operation in Critical Section

One possible correction is to place the operation in critical section. You can implement the critical section in multiple ways. For instance:

- You can place `var++` in a critical section. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The operation `var++` from the three tasks cannot interfere with each other.

  To implement the critical section, in the function `increment`, place the operation `var++` between calls to `begin_critical_section` and `end_critical_section`.

  ```
  int var;

  void begin_critical_section(void);
  void end_critical_section(void);

  void increment(void) {
        begin_critical_section();
        var++;
        end_critical_section();
  }

  void task1(void)  {
        increment();
  }

  void task2(void)  {
        increment();
  }

  void task3(void)  {
        increment();
  }
  ```
- You can place the call to `increment` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `increment` from the three tasks cannot interfere with each other.

  To implement the critical section, in each of the three tasks, call `increment` between calls to `begin_critical_section` and `end_critical_section`.

  ```
  int var;

  void begin_critical_section(void);
  void end_critical_section(void);

  void increment(void) {
        var++;
  }

  void task1(void)  {
  ```

```
        begin_critical_section();
        increment();
        end_critical_section();
   }

   void task2(void)  {
        begin_critical_section();
        increment();
        end_critical_section();
   }

   void task3(void)  {
        begin_critical_section();
        increment();
        end_critical_section();
   }
```

#### Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

| Option | Value |
|---|---|
| Temporally exclusive tasks (-temporal-exclusions-file) | task1 task2 task3 |

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

#### Example — Unprotected Operation in Threads Created with pthread_create

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    count = count + 1;
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    c = count;
    return NULL;
}
```

```
int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

In this example, Bug Finder detects the creation of separate threads with `pthread_create`. The **Data race** defect is raised because the operation `count = count + 1` in the thread with id `thread_increment` conflicts with the operation `c = count` in the thread with id `thread_get`. The variable `count` is accessed in multiple threads without a common protection.

The two conflicting operations are nonatomic. The operation `c = count` is nonatomic on 32-bit targets. See “Define Atomic Operations in Multitasking Code”.

#### Correction — Protect Operations with `pthread_mutex_lock` and `pthread_mutex_unlock` Pair

To prevent concurrent access on the variable `count`, protect operations on `count` with a critical section. Use the functions `pthread_mutex_lock` and `pthread_mutex_unlock` to implement the critical section.

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

### Data race on adjacent bit fields

#### Issue

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This issue occurs when:

1. Multiple tasks perform unprotected operations on bit fields that are part of the same structure.

   For instance, a task operates on field `errorFlag1` and another task on field `errorFlag2` in a variable of this type:

   ```
   struct errorFlags {
      unsigned int errorFlag1 : 1;
      unsigned int errorFlag2 : 1;
      ...
   }
   ```

   Suppose that the operations are not atomic with respect to each other. In other words, you have not implemented protection mechanisms to ensure that one operation is completed before another operation begins.
2. At least one of the unprotected operations is a write operation.

To find this defect, before analysis, you must specify the multitasking options. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see "Configuring Polyspace Multitasking Analysis Manually".

#### Risk

Adjacent bit fields that are part of the same structure might be stored in one byte in the same memory location. Read or write operations on all variables including bit fields occur one byte or word at a time. To modify only specific bits in a byte, steps similar to these steps occur in sequence:

1. The byte is loaded into RAM.
2. A mask is created so that only specific bits are modified to the intended value and the remaining bits remain unchanged.
3. A bitwise OR operation is performed between the copy of the byte in RAM and the mask.
4. The byte with specific bits modified is copied back from RAM.

When you access two different bit fields, these four steps have to be performed for each bit field. If the accesses are not protected, all four steps for one bit field might not be completed before the four steps for the other bit field begin. As a result, the modification of one bit field might undo the modification of an adjacent bit field. For instance, in the preceding example, the modification of `errorFlag1` and `errorFlag2` can occur in the following sequence. Steps 1,2 and 5 relate to modification of `errorFlag1` and while steps 3,4 and 6 relate to that of `errorFlag2`.

1. The byte with both `errorFlag1` and `errorFlag2` unmodified is copied into RAM, for purposes of modifying `errorFlag1`.
2. A mask that modifies only `errorFlag1` is bitwise OR-ed with this copy.
3. The byte containing both `errorFlag1` and `errorFlag2` unmodified is copied into RAM a second time, for purposes of modifying `errorFlag2`.
4. A mask that modifies only `errorFlag2` is bitwise OR-ed with this second copy.
5. The version with `errorFlag1` modified is copied back. This version has `errorFlag2` unmodified.
6. The version with `errorFlag2` modified is copied back. This version has `errorFlag1` unmodified and overwrites the previous modification.

#### Fix

To fix this defect, protect the operations on bit fields that are part of the same structure by using critical sections, temporal exclusion, or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon.

#### Example — Unprotected Operation on Global Variable from Multiple Tasks

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

In this example, `task1` and `task2` access different bit fields `IOFlag` and `SetupFlag`, which belong to the same structured variable `InterruptConfigbitsProc12`.

To emulate multitasking behavior, specify the options listed in this table.

| Option | Specification |
|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ |

| Option | Specification |
|---|---|
| **Tasks** on page 2-121 | task1<br>task2 |

At the command-line, use:

```
 polyspace-bug-finder
   -entry-points task1,task2
```

**Correction – Use Critical Sections**

One possible correction is to wrap the bit field access in a critical section. A critical section lies between a call to a lock function and an unlock function. In this correction, the critical section lies between the calls to functions `begin_critical_section` and `end_critical_section`.

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void begin_critical_section(void);
void end_critical_section(void);

void task1 (void) {
    begin_critical_section();
    InterruptConfigbitsProc12.IOFlag = 0;
    end_critical_section();
}

void task2 (void) {
    begin_critical_section();
    InterruptConfigbitsProc12.SetupFlag = 0;
    end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify options listed in this table.

| Option | Specification | |
|---|---|---|
| **Configure multitasking manually** on page 2-117 | ☑ | |
| **Tasks** on page 2-121 | task1<br>task2 | |
| **Critical section details** on page 2-133 | **Starting routine** | **Ending routine** |

| Option | Specification | |
|---|---|---|
| | begin_critical_section | end_critical_section |

At the command-line, use:

```
polyspace-bug-finder
  -entry-points task1,task2
  -critical-section-begin begin_critical_section:cs1
  -critical-section-end end_critical_section:cs1
```

#### Correction – Avoid Bit Fields

If you do not have memory constraints, use the `char` data type instead of bit fields. The `char` variables in a structure occupy at least one byte and do not have the thread-safety issues that come from bit manipulations in a byte-sized operation. Data races do not result from unprotected operations on different `char` variables that are part of the same structure.

```
typedef struct
{
   unsigned char IOFlag;
   unsigned char InterruptFlag;
   unsigned char Register1Flag;
   unsigned char SignFlag;
   unsigned char SetupFlag;
   unsigned char Register2Flag;
   unsigned char ProcessorFlag;
   unsigned char GeneralFlag;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

Though the checker does not flag this correction, do not use this correction for C99 or earlier. Only from C11 and later does the C Standard mandate that distinct `char` variables cannot be accessed using the same word.

#### Correction – Insert Bit Field of Size 0

You can enter a non-bit field member or an unnamed bit field member of size 0 between two adjacent bit fields that might be accessed concurrently. A non-bit field member or size 0 bit field member ensures that the subsequent bit field starts from a new memory location. In this corrected example, the size 0 bit field member ensures that `IOFlag` and `SetupFlag` are stored in distinct memory locations.

```
typedef struct
{
   unsigned int IOFlag :1;
   unsigned int InterruptFlag :1;
   unsigned int Register1Flag :1;
   unsigned int SignFlag :1;
   unsigned int : 0;
   unsigned int SetupFlag :1;
   unsigned int Register2Flag :1;
   unsigned int ProcessorFlag :1;
   unsigned int GeneralFlag :1;
} InterruptConfigbits_t;

InterruptConfigbits_t InterruptConfigbitsProc12;

void task1 (void) {
    InterruptConfigbitsProc12.IOFlag = 0;
}

void task2 (void) {
    InterruptConfigbitsProc12.SetupFlag = 0;
}
```

### Data race through standard library function call

#### Issue

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis".*

This issue occurs when:

1. Multiple tasks call the same standard library function.

   For instance, multiple tasks call the `strerror` function.
2. The calls are not protected using a common protection.

   For instance, the calls are not protected by the same critical section.

Functions flagged by this defect are not guaranteed to be reentrant. A function is reentrant if it can be interrupted and safely called again before its previous invocation completes execution. If a function is not reentrant, multiple tasks calling the function without protection can cause concurrency issues. For the list of functions that are flagged, see CON33-C: Avoid race conditions when using library functions.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see "Configuring Polyspace Multitasking Analysis Manually".

#### Risk

The functions flagged by this defect are nonreentrant because their implementations can use global or static variables. When multiple tasks call the function without protection, the function call from one task can interfere with the call from another task. The two invocations of the function can concurrently access the global or static variables and cause unpredictable results.

The calls can also cause more serious security vulnerabilities, such as abnormal termination, denial-of-service attack, and data integrity violations.

#### Fix

To fix this defect, do one of the following:

- Use a reentrant version of the standard library function if it exists.

  For instance, instead of `strerror()`, use `strerror_r()` or `strerror_s()`. For alternatives to functions flagged by this defect, see the documentation for CON33-C.
- Protect the function calls using common critical sections or temporal exclusion.

  See `Critical section details (-critical-section-begin -critical-section-end)` and `Temporally exclusive tasks (-temporal-exclusions-file)`.

  To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon. For an example, see below.

**Example — Unprotected Call to Standard Library Function from Multiple Tasks**

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Specification | |
|---|---|---|
| `Configure multitasking manually` | ☑ | |
| `Tasks (-entry-points)` | `task1`<br>`task2`<br>`task3` | |
| `Critical section details (-critical-section-begin -critical-section-end)` | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

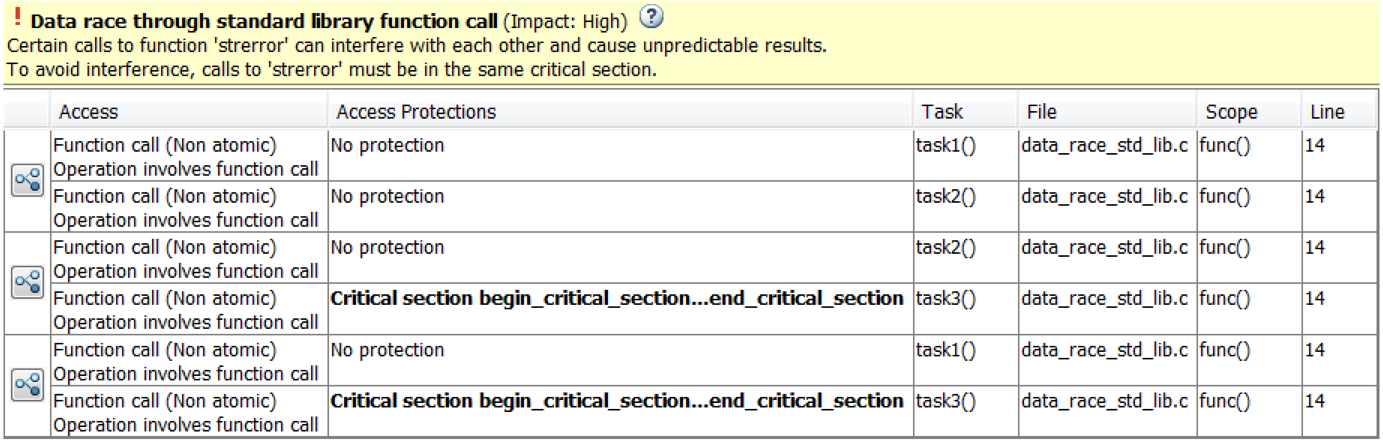
On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2,task3
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

In this example, the tasks, `task1`, `task2` and `task3`, call the function `func`. `func` calls the nonreentrant standard library function, `strerror`.

Though `task3` calls `func` inside a critical section, other tasks do not use the same critical section. Operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

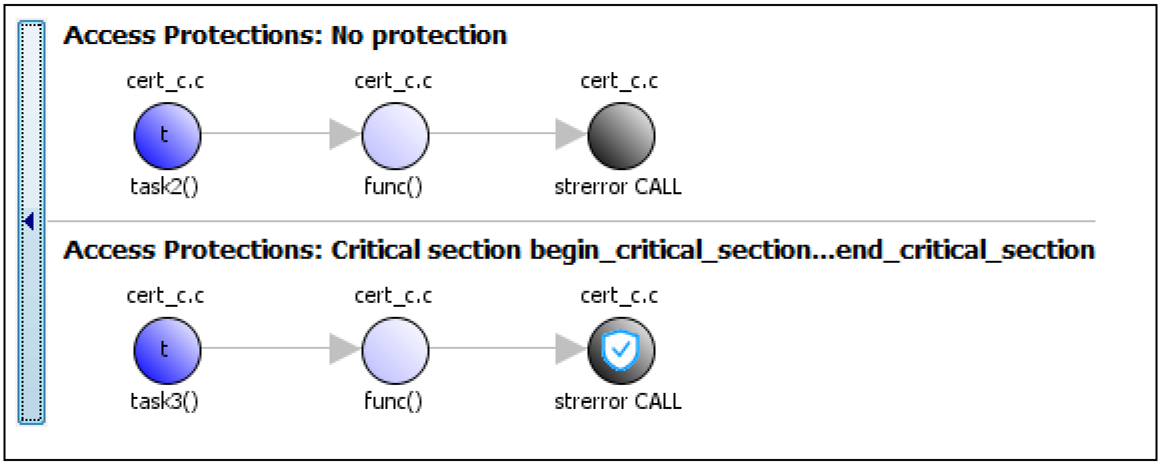
These three tasks are calling a nonreentrant standard library function without common protection. In your result details, you see each pair of conflicting function calls.



**! Data race through standard library function call** (Impact: High)
Certain calls to function 'strerror' can interfere with each other and cause unpredictable results.
To avoid interference, calls to 'strerror' must be in the same critical section.

| | Access | Access Protections | Task | File | Scope | Line |
|---|---|---|---|---|---|---|
| | Function call (Non atomic)<br>Operation involves function call | No protection | task1() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task2() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task2() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | **Critical section begin_critical_section...end_critical_section** | task3() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task1() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | **Critical section begin_critical_section...end_critical_section** | task3() | data_race_std_lib.c | func() | 14 |

If you click the icon, you see the function call sequence starting from the entry point to the standard library function call. You also see that the call starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.


Access Protections: No protection
cert_c.c
t
task2()
cert_c.c
func()
cert_c.c
strerror CALL
Access Protections: Critical section begin_critical_section...end_critical_section
cert_c.c
t
task3()
cert_c.c
func()
cert_c.c
strerror CALL

**Correction — Use Reentrant Version of Standard Library Function**

One possible correction is to use a reentrant version of the standard library function `strerror`. You can use the POSIX version `strerror_r` which has the same functionality but also guarantees thread-safety.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);
enum { BUFFERSIZE = 64 };

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char errmsg[BUFFERSIZE];
    if (strerror_r(errno, errmsg, BUFFERSIZE) != 0) {
      /* Handle error */
    }
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

**Correction — Place Function Call in Critical Section**

One possible correction is to place the call to `strerror` in critical section. You can implement the critical section in multiple ways.

For instance, you can place the call to the intermediate function `func` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `func` and therefore the calls to `strerror` from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call `func` between calls to `begin_critical_section` and `end_critical_section`.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    begin_critical_section();
    func(fptr1);
    end_critical_section();
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     begin_critical_section();
     func(fptr2);
     end_critical_section();
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

#### Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

| Option | Value |
|---|---|
| `Temporally exclusive tasks (-temporal-exclusions-file)` | `task1 task2 task3` |

On the command-line, you can use the following:

```
polyspace-bug-finder
    -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

## Check Information

**Category:** Concurrency Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-366

# CWE Rule 369

Divide By Zero

## Description

### Rule Description

*The product divides a value by zero.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Float division by zero**
- **Integer division by zero**
- **Tainted division operand**
- **Tainted modulo operand**

## Examples

### Float division by zero

#### Issue

This issue occurs when the denominator of a division operation can be a zero-valued floating point number.

#### Risk

A division by zero can result in a program crash.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, for instance, when you handle infinities in your code, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

By default, a Bug Finder analysis does not recognize infinities and `NaNs`. Operations that results in infinities and `NaNs` might be flagged as defects. To handle infinities and `NaN` values in your code, use the option `Consider non finite floats (-allow-non-finite-floats).`

**Extend Checker**

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

**Example — Dividing a Floating Point Number by Zero**

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

**Correction — Check Before Division**

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    if( ((int)denom) != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

**Correction — Change Denominator**

One possible correction is to change the denominator value so that `denom` is not zero.

```
float fraction(float num)
{
    float denom = 2.0;
    float result = 0.0;
    result = num/denom;

    return result;
}
```

### Integer division by zero

#### Issue

This issue occurs when the denominator of a division or modulo operation can be a zero-valued integer.

#### Risk

A division by zero can result in a program crash.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

It is a good practice to check for zero values of a denominator before division and handle the error. Instead of performing the division directly:

```
res = num/den;
```

use a library function that handles zero values of the denominator before performing the division:

```
res = div(num, den);
```

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;
    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

#### Correction — Check Before Division

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

#### Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2;
    int result = 0;

    result = num/denom;

    return result;
}
```

#### Example — Modulo Operation with Zero

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        arr[i] = input % i;
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

In this example, Polyspace flags the modulo operation as a division by zero. Because modulo is inherently a division operation, the divisor (right hand argument) cannot be zero. The modulo operation uses the `for` loop index as the divisor. However, the `for` loop starts at zero, which cannot be an iterator.

#### Correction — Check Divisor Before Operation

One possible correction is checking the divisor before the modulo operation. In this example, see if the index `i` is zero before the modulo operation.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        if(i != 0)
        {
             arr[i] = input % i;
        }
        else
        {
             arr[i] = input;
        }
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

**Correction — Change Divisor**

Another possible correction is changing the divisor to a nonzero integer. In this example, add one to the index before the % operation to avoid dividing by zero.

```
int mod_arr(int input)
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
         arr[i] = input % (i+1);
    }

    return arr[0]+arr[1]+arr[2]+arr[3]+arr[4];
}
```

### Tainted division operand

**Issue**

This issue occurs when one or both integer operands in a division operation comes from unsecure sources.

**Risk**

- If the numerator is the minimum possible value and the denominator is `-1`, your division operation overflows because the result cannot be represented by the current variable size.
- If the denominator is zero, your division operation fails possibly causing your program to crash.

These risks can be used to execute arbitrary code. This code is usually outside the scope of a program's implicit security policy.

**Fix**

Before performing the division, validate the values of the operands. Check for denominators of `0` or `-1`, and numerators of the minimum integer value.

**Extend Checker**

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of

Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Division of Function Arguments

```
#include <limits.h>
#include <stdio.h>

extern void print_int(int);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    int r =  num/denum; //Noncompliant
    print_int(r);
    return r;
}
```

This example function divides two argument variables, then prints and returns the result. The argument values are unknown and can cause division by zero or integer overflow.

#### Correction — Check Values

One possible correction is to check the values of the numerator and denominator before performing the division.

```
#include <limits.h>
#include <stdio.h>

extern void print_long(long);

int taintedintdivision(void) {
    long num, denum;
    scanf("%lf %lf", &num, &denum);
    long res= 0;
    if (denum!=0 && !(num==INT_MIN && denum==-1)) {
        res = num/denum;
    }
    print_long(res);
    return res;
}
```

### Tainted modulo operand

#### Issue

This issue occurs when one or both integer operands in a remainder operation (`%`) comes from unsecure sources.

#### Risk

- If the second remainder operand is zero, your remainder operation fails, causing your program to crash.
- If the second remainder operand is `-1`, your remainder operation can overflow if the remainder operation is implemented based on the division operation that can overflow.
- If one of the operands is negative, the operation result is uncertain. For C89, the modulo operation is not standardized, so the result from negative operands is implementation-defined.

These risks can be exploited by attackers to gain access to your program or the target in general.

#### Fix

Before performing the modulo operation, validate the values of the operands. Check the second operand for values of `0` and `-1`. Check both operands for negative values.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Modulo with User Input

```
#include <stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem =  128%userden; //Noncompliant
    print_int(rem);
    return rem;
}
```

In this example, the function performs a modulo operation by using a user input. The input is not checked before calculating the remainder for values that can crash the program, such as 0 and -1.

#### Correction — Check Operand Values

One possible correction is to check the values of the operands before performing the modulo operation. In this corrected example, the modulo operation continues only if the second operand is greater than zero.

```
#include<stdio.h>
extern void print_int(int);

int taintedintmod(void) {
    int userden;
    scanf("%d", &userden);
    int rem = 0;
    if (userden > 0 ) {
        rem = 128 % userden;
    }
    print_int(rem);
    return rem;
}
```

## Check Information

**Category:** Numeric Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

**Topics**

“Check for and Review Coding Standard Violations”

**External Websites**

CWE-369

# CWE Rule 375

Returning a Mutable Object to an Untrusted Caller

## Description

### Rule Description

*Sending non-cloned mutable data as a return value may result in that data being altered or deleted by the calling function.*

### Polyspace Implementation

The rule checker checks for **Return of non-const handle to encapsulated data member**.

## Examples

### Return of non-const handle to encapsulated data member

#### Issue

This issue occurs when:

- A class method returns a handle to a data member. Handles include pointers and references.
- The method is more accessible than the data member. For instance, the method has access specifier `public`, but the data member is `private` or `protected`.

#### Risk

The access specifier determines the accessibility of a class member. For instance, a class member declared with the `private` access specifier cannot be accessed outside a class. Therefore, nonmember, nonfriend functions cannot modify the member.

When a class method returns a handle to a less accessible data member, the member accessibility changes. For instance, if a `public` method returns a pointer to a `private` data member, the data member is effectively not `private` anymore. A nonmember, nonfriend function calling the `public` method can use the returned pointer to view and modify the data member.

Also, if you assign the pointer to a data member of an object to another pointer, when you delete the object, the second pointer can be left dangling. The second pointer points to the part of an object that does not exist anymore.

#### Fix

One possible fix is to avoid returning a handle to a data member from a class method. Return a data member by value so that a copy of the member is returned. Modifying the copy does not change the data member.

If you must return a handle, use a `const` qualifier with the method return type so that the handle allows viewing, but not modifying, the data member.

#### Example — Return of Pointer to private Data Member

```
#include <string>
#define NUM_RECORDS 100
```

```
struct Date {
    int dd;
    int mm;
    int yyyy;
};


struct Period {
    Date startDate;
    Date endDate;
};

class DataBaseEntry {
private:
    std::string employeeName;
    Period employmentPeriod;
public:
    Period* getPeriod(void);
};

Period* DataBaseEntry::getPeriod(void) {
    return &employmentPeriod;
}


void use(Period*);
void reset(Period*);

int main() {
    DataBaseEntry dataBase[NUM_RECORDS];
    Period* tempPeriod;
    for(int i=0;i < NUM_RECORDS;i++) {
        tempPeriod = dataBase[i].getPeriod();
        use(tempPeriod);
        reset(tempPeriod);
    }
    return 0;
}

void reset(Period* aPeriod) {
       aPeriod->startDate.dd = 1;
       aPeriod->startDate.mm = 1;
       aPeriod->startDate.yyyy = 2000;
}
```

In this example, `employmentPeriod` is `private` to the class `DataBaseEntry`. It is therefore immune from modification by nonmember, nonfriend functions. However, returning a pointer to `employmentPeriod` breaks this encapsulation. For instance, the nonmember function `reset` modifies the member `startDate` of `employmentPeriod`.

#### Correction: Return Member by Value

One possible correction is to return the data member `employmentPeriod` by value instead of pointer. Modifying the return value does not change the data member because the return value is a copy of the data member.

```
#include <string>
#define NUM_RECORDS 100

struct Date {
    int dd;
    int mm;
    int yyyy;
};


struct Period {
    Date startDate;
    Date endDate;
};

class DataBaseEntry {
private:
    std::string employeeName;
    Period employmentPeriod;
public:
    Period getPeriod(void);
};

Period DataBaseEntry::getPeriod(void) {
    return employmentPeriod;
}


void use(Period*);
void reset(Period*);

int main() {
    DataBaseEntry dataBase[NUM_RECORDS];
    Period tempPeriodVal;
    Period* tempPeriod;
    for(int i=0;i < NUM_RECORDS;i++) {
        tempPeriodVal = dataBase[i].getPeriod();
        tempPeriod = &tempPeriodVal;
        use(tempPeriod);
        reset(tempPeriod);
    }
    return 0;
}

void reset(Period* aPeriod) {
       aPeriod->startDate.dd = 1;
       aPeriod->startDate.mm = 1;
       aPeriod->startDate.yyyy = 2000;
}
```

## Check Information

**Category:** State Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

CWE-375

# CWE Rule 396

Declaration of Catch for Generic Exception

## Description

### Rule Description

*Catching overly broad exceptions promotes complex error handling code that is more likely to contain security vulnerabilities.*

### Polyspace Implementation

The rule checker checks for **Declaration of catch for generic exception**.

## Examples

### Declaration of catch for generic exception

#### Issue

This issue occurs when you design `catch` blocks to handle generic high level exceptions. Polyspace checks the exception specifications of a function and raises a violation if a catch block handles:

- All exceptions by using a catch-all block.
- Exception of the class `std::exception`. Because `std::exception` is the base class for all standard C++ exceptions, a catch block that handles `std::exception` type matches all derived exceptions.

#### Risk

Using generic catch blocks hides the emergence of unexpected new exceptions and hinders their proper handling. Such generic catch blocks makes the code vulnerable to security issues. For instance:

```
void foo(void){
    try{
        //..
    }catch(std::exception& e){
        //Log error
    }
}
```

Because the `catch` block handles the generic `std::exception` class, this code hides unexpected exceptions or those that need to be handled differently. For instance, If an unexpected `std::invalid_argument` exception is raised in `foo()`, it might remain undetected by the developer because it is matched with the catch block. Because the catch block is not programmed to handle `std::invalid_argument` properly, the poorly handled exception becomes a vulnerability for the code.

#### Fix

To fix this defect, avoid catching high-level generic exceptions. Write catch blocks that handle specific exceptions to enable handling different exceptions in different ways. Unexpected or new exceptions are also easily detected when catch blocks are specific.

#### Example — Avoid Generic Catch Blocks

```
#include<string>
#include<exception>
#include<stdexcept>

extern void logger(std::string s, std::exception& e);
void foo(void){
    try{
        //..
    }catch(std::exception& e){
         logger("foo failed", e);
    }
}
```

In this example, Polyspace flags the use of the generic `catch` block.

#### Correction — Use Specific Catch Blocks

To resolve this defect, use specific catch blocks. When you use specific catch blocks, new and unexpected exceptions become unhandled exceptions that are easily detected.

```
#include<string>
#include<exception>
extern void logger(std::string s, std::exception& e);
void foo(void){
    try{
        //..
    }catch(std::domain_error& e){
         logger("Domain Error in foo", e);
    }catch(std::invalid_argument& e){
        logger("Invalid Argument in foo", e);
    }
}
```

## Check Information

**Category:** Error Conditions, Return Values, Status Codes

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

#### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-396

# CWE Rule 397

Declaration of Throws for Generic Exception

## Description

### Rule Description

*Throwing overly broad exceptions promotes complex error handling code that is more likely to contain security vulnerabilities.*

### Polyspace Implementation

The rule checker checks for **Declaration of throw for generic exception**.

## Examples

### Declaration of throw for generic exception

#### Issue

This issue occurs when a function throws a generic exception. Polyspace checks the throw statements, dynamic exception specifications, or the documentation of a function and raises this defect if any of these conditions is true:

- The function contains `throw` statements that raise generic exceptions.
- The dynamic exception specification contains generic exceptions.
- The function is documented with a `@throw` comment that contains generic exceptions.

Polyspace considers exceptions of the class `std::exception` as generic.These are base exception classes that have many derived exception classes.

#### Risk

Raising generic exceptions hinders proper error handling and error recovery. For instance:

```
void foo(void) throw(std::exception){
    //...
}

void caller(void){
    try{
        //...
        foo();
    }catch(std::exception& e){
        // Need to figure out what the error is.
    }
}
```

When handling the exceptions raised by `foo()`, determining what errors might happen in `foo()` is handled by `caller()`. Because the exception specification of `foo()` is broad, the emergence of new or unexpected exceptions might not be detected. Handling and recovering from errors in `foo()` becomes more complicated for `caller()`.

### Fix

To resolve this defect, use specific exceptions when documenting the exception in a function. Caller functions are then able to address specific exceptions by using specific error-handling code. Unexpected exceptions are easily detected.

#### Example — Avoid Generic Exception Specification

```
#include<exception>
#include<stdexcept>
#include<iostream>

void foo(void) throw(std::exception){
    //...
}
// @throw std::exception
void bar() {  //Noncompliant
    //...
    throw std::exception();
}

void foo1(){
    //...
    throw std::exception();
}
int main(void){
    try{
        //...
        foo();
    }catch(std::exception& e){
        // Need to figure out what the error is.

    }
    try{
        //...
        bar();
    }catch(std::exception& e){
        // Need to figure out what the error is.

    }
    try{
        //...
        foo1();
    }catch(std::exception& e){
        // Need to figure out what the error is.

    }
}
```

In this example, Polyspace flags the generic `throw` specifications. For instance:

- Polyspace flags the dynamic exception specification of `foo()` because it uses `std::exception`.
- Polyspace flags the function `bar()` because it is documented to raise the generic exception `std::exception`. Polyspace also flags the generic `throw` statement in this function.
- Polyspace flags the generic `throw` statement in `foo1()`.

Because these functions raise generic exceptions, identifying the exceptions is handled by the caller function `main()`. Because the caller does not know which specific exception are expected, it might fail to detect unexpected exceptions.

**Correction — Use Specific Exceptions**

To fix this issue, document specific exceptions that might arise from a function. This specific documentation enables the caller function to address specific exceptions. Unexpected exceptions become unhandled exceptions and are easily detected.

```
#include<exception>
#include<stdexcept>
#include<iostream>

void foo(void) throw(std::overflow_error){
    //...
}
// @throw std::invalid_argument
void bar() {
    //...
}

void foo1(){
    //...
    throw std::range_error("MSG");
}
int main(void){
    try{
        //...
        foo();
    }catch(std::overflow_error& e){
        // Expecting an overflow.

    }
    try{
        //...
        bar();
    }catch(std::invalid_argument& e){
        // Expecting an invalid argument.

    }
    try{
        //...
        foo1();
    }catch(std::range_error& e){
        // Expecting a range error.

    }
}
```

## Check Information

**Category:** Error Conditions, Return Values, Status Codes

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

CWE-397

# CWE Rule 401

Missing Release of Memory after Effective Lifetime

## Description

### Rule Description

*The software does not sufficiently track and release allocated memory after it has been used, which slowly consumes remaining memory.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Memory leak**
- **Thread-specific memory leak**

## Examples

### Memory leak

#### Issue

This issue occurs when you do not free a block of memory allocated through `malloc`, `calloc`, `realloc`, or `new`. If the memory is allocated in a function, the defect does not occur if:

- Within the function, you free the memory using `free` or `delete`.
- The function returns the pointer assigned by `malloc`, `calloc`, `realloc`, or `new`.
- The function stores the pointer in a global variable or in a parameter.

#### Risk

Dynamic memory allocation functions such as `malloc` allocate memory on the heap. If you do not release the memory after use, you reduce the amount of memory available for another allocation. On embedded systems with limited memory, you might end up exhausting available heap memory even during program execution.

#### Fix

Determine the scope where the dynamically allocated memory is accessed. Free the memory block at the end of this scope.

To free a block of memory, use the `free` function on the pointer that was used during memory allocation. For instance:

```
ptr = (int*)malloc(sizeof(int));
...
free(ptr);
```

It is a good practice to allocate and free memory in the same module at the same level of abstraction. For instance, in this example, `func` allocates and frees memory at the same level but `func2` does not.

```
void func() {
  ptr = (int*)malloc(sizeof(int));
  {
    ...
  }
  free(ptr);
}

void func2() {
  {
   ptr = (int*)malloc(sizeof(int));
   ...
  }
  free(ptr);
}
```

See CERT-C Rule MEM00-C.

**Example — Dynamic Memory Not Released Before End of Function**

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
}
```

In this example, `pi` is dynamically allocated by `malloc`. The function `assign_memory` does not free the memory, nor does it return `pi`.

**Correction — Free Memory**

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

```
#include<stdlib.h>
#include<stdio.h>

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
    free(pi);
}
```

**Correction — Return Pointer from Dynamic Allocation**

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

```
#include<stdlib.h>
#include<stdio.h>

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
            printf("Memory allocation failed");
            return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

**Example — Memory Leak with New/Delete**

```
#define NULL '\0'

void initialize_arr1(void)
{
    int *p_scalar = new int(5);
}

void initialize_arr2(void)
{
    int *p_array = new int[5];
}
```

In this example, the functions create two variables, `p_scalar` and `p_array`, using the `new` keyword. However, the functions end without cleaning up the memory for these pointers. Because the functions used `new` to create these variables, you must clean up their memory by calling `delete` at the end of each function.

**Correction — Add Delete**

To correct this error, add a `delete` statement for every `new` initialization. If you used brackets `[]` to instantiate a variable, you must call delete with brackets as well.

```
#define NULL '\0'

void initialize_arrs(void)
{
    int *p_scalar = new int(5);
    int *p_array = new int[5];

    delete p_scalar;
    p_scalar = NULL;

    delete[] p_array;
    p_scalar = NULL;
}
```

### Thread-specific memory leak

#### Issue

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

This issue occurs when you do not free thread-specific dynamically allocated memory before the end of a thread.

To create thread-specific storage, you generally do these steps:

1. You create a key for thread-specific storage.
2. You create the threads.
3. In each thread, you allocate storage dynamically and then associate the key with this storage.

   After the association, you can read the stored data later using the key.
4. Before the end of the thread, you free the thread-specific memory using the key.

The checker flags execution paths in the thread where the last step is missing.

The checker works on these families of functions:

- `tss_get` and `tss_set` (C11)
- `pthread_getspecific` and `pthread_setspecific` (POSIX)

#### Risk

The data stored in the memory is available to other processes even after the threads end (memory leak). Besides security vulnerabilities, memory leaks can shrink the amount of available memory and reduce performance.

#### Fix

Free dynamically allocated memory before the end of a thread.

You can explicitly free dynamically allocated memory with functions such as `free`.

Alternatively, when you create a key, you can associate a destructor function with the key. The destructor function is called with the key value as argument at the end of a thread. In the body of the destructor function, you can free any memory associated with the key. If you use this method, Bug Finder still flags a defect. Ignore this defect with appropriate comments. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example — Memory Not Freed at End of Thread**

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;

  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);

  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */
  }
  print_data();
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }

  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

In this example, the start function of each thread `func` calls two functions:

- `add_data`: This function allocates storage dynamically and associates the storage with a key using the `tss_set` function.
- `print_data`: This function reads the stored data using the `tss_get` function.

At the points where `func` returns, the dynamically allocated storage has not been freed.

**Correction — Free Dynamically Allocated Memory Explicitly**

One possible correction is to free dynamically allocated memory explicitly before leaving the start function of a thread. See the highlighted change in the corrected version.

In this corrected version, a defect still appears on the `return` statement in the error handling section of `func`. The defect cannot occur in practice because the error handling section is entered only if dynamic memory allocation fails. Ignore this remaining defect with appropriate comments. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

```
#include <threads.h>
#include <stdlib.h>

/* Global key to the thread-specific storage */
tss_t key;
enum { MAX_THREADS = 3 };


int add_data(void) {
  int *data = (int *)malloc(2 * sizeof(int));
  if (data == NULL) {
    return -1;  /* Report error  */
  }
  data[0] = 0;
  data[1] = 1;

  if (thrd_success != tss_set(key, (void *)data)) {
    /* Handle error */
  }
  return 0;
}

void print_data(void) {
  /* Get this thread's global data from key */
  int *data = tss_get(key);
  if (data != NULL) {
    /* Print data */
  }
}

int func(void *dummy) {
  if (add_data() != 0) {
    return -1;  /* Report error */
  }
  print_data();
  free(tss_get(key));
  return 0;
}

int main(void) {
  thrd_t thread_id[MAX_THREADS];

  /* Create the key before creating the threads */
  if (thrd_success != tss_create(&key, NULL)) {
    /* Handle error */
  }

  /* Create threads that would store specific storage */
  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_create(&thread_id[i], func, NULL)) {
      /* Handle error */
    }
  }

  for (size_t i = 0; i < MAX_THREADS; i++) {
    if (thrd_success != thrd_join(thread_id[i], NULL)) {
      /* Handle error */
    }
  }

  tss_delete(key);
  return 0;
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

Check CWE (-cwe)

### Topics

“Check for and Review Coding Standard Violations”

**External Websites**
CWE-401

# CWE Rule 413

Improper Resource Locking

## Description

### Rule Description

*The software does not lock or does not correctly lock a resource when the software must have exclusive access to the resource.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Data race**
- **Data race through standard library function call**
- **Shared data access within signal handler**

## Examples

### Data race

#### Issue

*This checker is deactivated in a default Polyspace* as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Analysis"*.

This issue occurs when:

1. Multiple tasks perform unprotected operations on a shared variable.
2. At least one task performs a write operation.
3. At least one operation is nonatomic. To detect data race on both atomic and nonatomic operations, use the options `-detect-atomic-data-race`. See "Extend Data Race Checkers to Atomic Operations".

   See "Define Atomic Operations in Multitasking Code".

If you activate this checker without specifying the multitasking options first, you see a warning in the log:To find this defect, specify the multitasking options before analysis.

To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see "Configuring Polyspace Multitasking Analysis Manually".

#### Risk

Data race can result in unpredictable values of the shared variable because you do not control the order of the operations in different tasks.

Data races between two write operations are more serious than data races between a write and read operation. Two write operations can interfere with each other and result in indeterminate values. To identify write-write conflicts, use the filters on the **Detail** column of the **Results List** pane. For these conflicts, the **Detail** column shows the additional line:

```
 Variable value may be altered by write-write concurrent access.
```

See also “Filter and Group Results in Polyspace Desktop User Interface” or “Filter and Sort Results in Polyspace Access Web Interface”.

#### Fix

To fix this defect, protect the operations on the shared variable using critical sections, temporal exclusion or another means. See “Protections for Shared Variables in Multitasking Code”.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon. For an example, see below.

#### Extend Checker

Extend this checker to check for data races in operations that Bug Finder might not detect by default. For instance:

- You might be using multithreading functions that are not supported by Polyspace. Extend this checker by mapping the functions of your multithreading functions to their known POSIX equivalent. See “Extend Concurrency Defect Checkers to Unsupported Multithreading Environments”.
- Polyspace assumes that certain operations are atomic and excludes them from data race checks. See “Define Atomic Operations in Multitasking Code”. These assumptions might not apply to your environment. To extend the data race checkers to include these operations, use the option `-detect-atomic-data-race`. See “Extend Data Race Checkers to Atomic Operations”.

#### Example — Unprotected Operation on Global Variable from Multiple Tasks

```
int var;
void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
    var++;
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
     begin_critical_section();
     increment();
    end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Specification | |
|---|---|---|
| `Configure multitasking manually` | ☑ | |
| `Tasks (-entry-points)` | `task1`<br>`task2`<br>`task3` | |
| `Critical section details (-critical-section-begin -critical-section-end)` | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2,task3
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

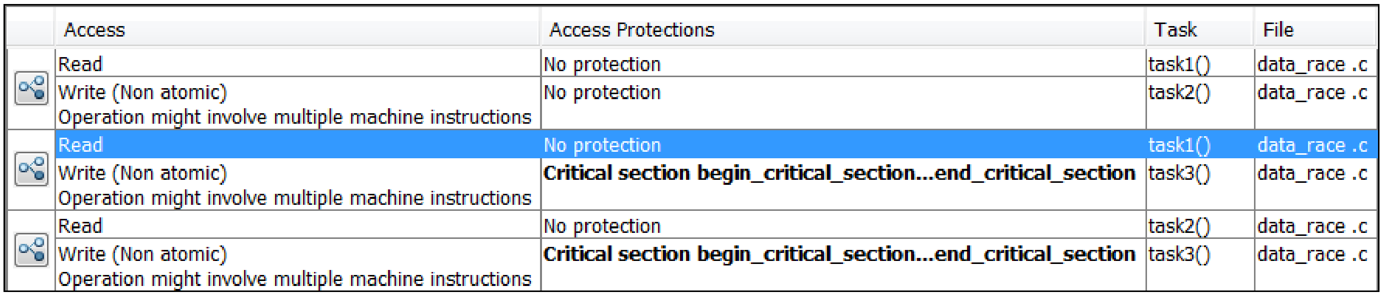
In this example, the tasks `task1`, `task2`, and `task3` call the function `increment`. `increment` contains the operation `var++` that can involve multiple machine instructions including:

- Reading `var`.
- Writing an increased value to `var`.

These machine instructions, when executed from `task1` and `task2`, can occur concurrently in an unpredictable sequence. For example, reading `var` from `task1` can occur either before or after writing to `var` from `task2`. Therefore the value of `var` can be unpredictable.

Though `task3` calls `increment` inside a critical section, other tasks do not use the same critical section. The operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

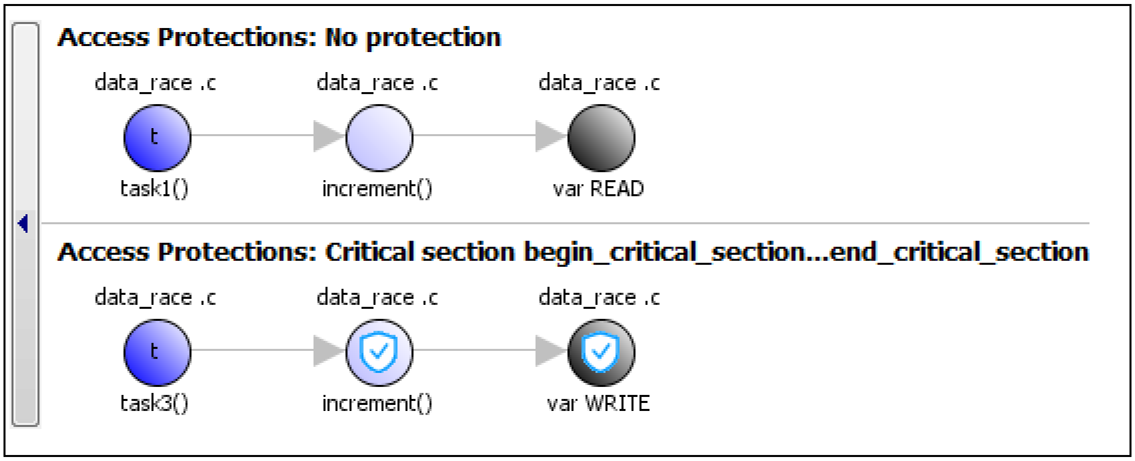
Therefore, the three tasks are operating on a shared variable without common protection. In your result details, you see each pair of conflicting function calls.



| | Access | Access Protections | Task | File |
|---|---|---|---|---|
| | Read | No protection | task1() | data_race .c |
| | Write (Non atomic)<br>Operation might involve multiple machine instructions | No protection | task2() | data_race .c |
| | Read | No protection | task1() | data_race .c |
| | Write (Non atomic)<br>Operation might involve multiple machine instructions | **Critical section begin_critical_section...end_critical_section** | task3() | data_race .c |
| | Read | No protection | task2() | data_race .c |
| | Write (Non atomic)<br>Operation might involve multiple machine instructions | **Critical section begin_critical_section...end_critical_section** | task3() | data_race .c |

If you click the icon, you see the function call sequence starting from the entry point to the read or write operation. You also see that the operation starting from `task3` is in a critical section. The

**Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.


Access Protections: No protection

Access Protections: Critical section begin_critical_section...end_critical_section

**Correction — Place Operation in Critical Section**

One possible correction is to place the operation in critical section. You can implement the critical section in multiple ways. For instance:

- You can place `var++` in a critical section. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The operation `var++` from the three tasks cannot interfere with each other.

  To implement the critical section, in the function `increment`, place the operation `var++` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      begin_critical_section();
      var++;
      end_critical_section();
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
      increment();
}
```

- You can place the call to `increment` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `increment` from the three tasks cannot interfere with each other.

  To implement the critical section, in each of the three tasks, call `increment` between calls to `begin_critical_section` and `end_critical_section`.

```
int var;

void begin_critical_section(void);
void end_critical_section(void);

void increment(void) {
      var++;
}

void task1(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task2(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

**Correction — Make Tasks Temporally Exclusive**

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

| Option | Value |
|---|---|
| Temporally exclusive tasks (-temporal-exclusions-file) | task1 task2 task3 |

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

#### Example — Unprotected Operation in Threads Created with pthread_create

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    count = count + 1;
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    c = count;
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

In this example, Bug Finder detects the creation of separate threads with `pthread_create`. The **Data race** defect is raised because the operation `count = count + 1` in the thread with id `thread_increment` conflicts with the operation `c = count` in the thread with id `thread_get`. The variable `count` is accessed in multiple threads without a common protection.

The two conflicting operations are nonatomic. The operation `c = count` is nonatomic on 32-bit targets. See “Define Atomic Operations in Multitasking Code”.

#### Correction — Protect Operations with `pthread_mutex_lock` and `pthread_mutex_unlock` Pair

To prevent concurrent access on the variable `count`, protect operations on `count` with a critical section. Use the functions `pthread_mutex_lock` and `pthread_mutex_unlock` to implement the critical section.

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;


void* increment_count(void* args)
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

void* set_count(void *args)
{
    long long c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return NULL;
}

int main(void)
{
    pthread_t thread_increment;
    pthread_t thread_get;

    pthread_create(&thread_increment, NULL, increment_count, NULL);
    pthread_create(&thread_get, NULL, set_count, NULL);

    pthread_join(thread_get, NULL);
    pthread_join(thread_increment, NULL);

    return 1;
}
```

### Data race through standard library function call

#### Issue

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

This issue occurs when:

1. Multiple tasks call the same standard library function.

   For instance, multiple tasks call the `strerror` function.
2. The calls are not protected using a common protection.

   For instance, the calls are not protected by the same critical section.

Functions flagged by this defect are not guaranteed to be reentrant. A function is reentrant if it can be interrupted and safely called again before its previous invocation completes execution. If a function is not reentrant, multiple tasks calling the function without protection can cause concurrency issues. For the list of functions that are flagged, see CON33-C: Avoid race conditions when using library functions.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

#### Risk

The functions flagged by this defect are nonreentrant because their implementations can use global or static variables. When multiple tasks call the function without protection, the function call from one task can interfere with the call from another task. The two invocations of the function can concurrently access the global or static variables and cause unpredictable results.

The calls can also cause more serious security vulnerabilities, such as abnormal termination, denial-of-service attack, and data integrity violations.

**Fix**

To fix this defect, do one of the following:

- Use a reentrant version of the standard library function if it exists.

  For instance, instead of `strerror()`, use `strerror_r()` or `strerror_s()`. For alternatives to functions flagged by this defect, see the documentation for CON33-C.
- Protect the function calls using common critical sections or temporal exclusion.

  See `Critical section details (-critical-section-begin -critical-section-end)` and `Temporally exclusive tasks (-temporal-exclusions-file)`.

  To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon. For an example, see below.

**Example — Unprotected Call to Standard Library Function from Multiple Tasks**

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Specification | |
|---|---|---|
| `Configure multitasking manually` | ☑ | |
| `Tasks (-entry-points)` | `task1`<br>`task2`<br>`task3` | |
| `Critical section details (-critical-section-begin -critical-section-end)` | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

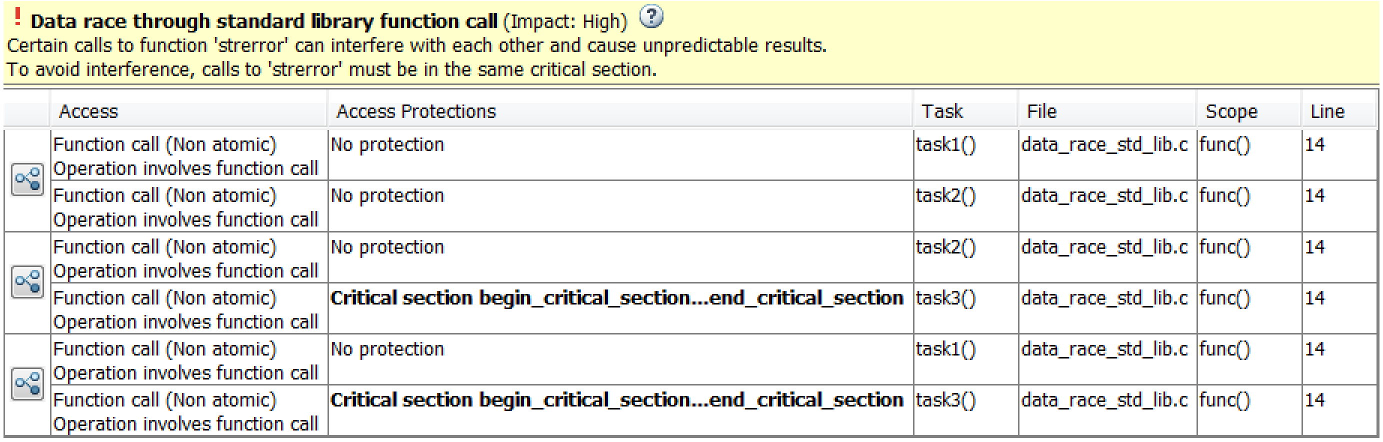
On the command-line, you can use the following:

```
 polyspace-bug-finder
   -entry-points task1,task2,task3
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

In this example, the tasks, `task1`, `task2` and `task3`, call the function `func`. `func` calls the nonreentrant standard library function, `strerror`.

Though `task3` calls `func` inside a critical section, other tasks do not use the same critical section. Operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

These three tasks are calling a nonreentrant standard library function without common protection. In your result details, you see each pair of conflicting function calls.

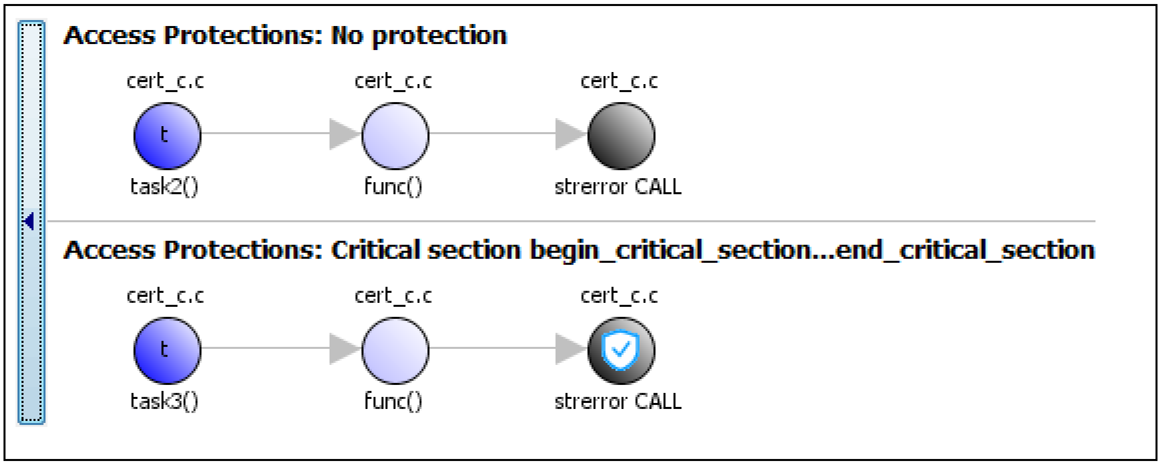


! **Data race through standard library function call** (Impact: High)
Certain calls to function 'strerror' can interfere with each other and cause unpredictable results.
To avoid interference, calls to 'strerror' must be in the same critical section.

| | Access | Access Protections | Task | File | Scope | Line |
|---|---|---|---|---|---|---|
| | Function call (Non atomic)<br>Operation involves function call | No protection | task1() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task2() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task2() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | **Critical section begin_critical_section...end_critical_section** | task3() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task1() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | **Critical section begin_critical_section...end_critical_section** | task3() | data_race_std_lib.c | func() | 14 |

If you click the icon, you see the function call sequence starting from the entry point to the standard library function call. You also see that the call starting from `task3` is in a critical section.

The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.


Access Protections: No protection
cert_c.c
cert_c.c
cert_c.c
t
task2()
func()
strerror CALL
Access Protections: Critical section begin_critical_section...end_critical_section
cert_c.c
cert_c.c
cert_c.c
t
task3()
func()
strerror CALL

**Correction — Use Reentrant Version of Standard Library Function**

One possible correction is to use a reentrant version of the standard library function `strerror`. You can use the POSIX version `strerror_r` which has the same functionality but also guarantees thread-safety.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);
enum { BUFFERSIZE = 64 };

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char errmsg[BUFFERSIZE];
    if (strerror_r(errno, errmsg, BUFFERSIZE) != 0) {
      /* Handle error */
    }
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

#### Correction — Place Function Call in Critical Section

One possible correction is to place the call to `strerror` in critical section. You can implement the critical section in multiple ways.

For instance, you can place the call to the intermediate function `func` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `func` and therefore the calls to `strerror` from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call `func` between calls to `begin_critical_section` and `end_critical_section`.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    begin_critical_section();
    func(fptr1);
    end_critical_section();
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     begin_critical_section();
     func(fptr2);
     end_critical_section();
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

#### Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

| Option | Value |
|---|---|
| Temporally exclusive tasks (-temporal-exclusions-file) | task1 task2 task3 |

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

### Shared data access within signal handler

#### Issue

This issue occurs when you access or modify a shared object inside a signal handler.

#### Risk

When you define a signal handler function to access or modify a shared object, the handler accesses or modifies the shared object when it receives a signal. If another function is already accessing the shared object, that function causes a race condition and can leave the data in an inconsistent state.

#### Fix

To access or modify shared objects inside a signal handler, check that the objects are lock-free atomic, or, if they are integers, declare them as `volatile sig_atomic_t`.

#### Example — int Variable Access in Signal Handler

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* declare global variable. */
int e_flag;

void sig_handler(int signum)
{
    /* Signal handler accesses variable that is not
    of type volatile sig_atomic_t. */
    e_flag = signum;
}
```

```
int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` accesses `e_flag`, a variable of type `int`. A concurrent access by another function can leave `e_flag` in an inconsistent state.

**Correction — Declare Variable of Type `volatile sig_atomic_t`**

Before you access a shared variable from a signal handler, declare the variable with type `volatile sig_atomic_t` instead of `int`. You can safely access variables of this type asynchronously.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

/* Declare variable of type volatile sig_atomic_t. */
volatile sig_atomic_t e_flag;
void sig_handler(int signum)
{
    /* Use variable of proper type inside signal handler. */
    e_flag = signum;

}

int func(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
        abort();
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
        abort();
    }
    /* More code */
    return 0;
}
```

## Check Information

**Category:** Resource Locking Problems

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-413

# CWE Rule 415

Double Free

## Description

### Rule Description

*The product calls free() twice on the same memory address, potentially leading to modification of unexpected memory locations.*

### Polyspace Implementation

The rule checker checks for **Deallocation of previously deallocated pointer**.

## Examples

### Deallocation of previously deallocated pointer

#### Issue

This issue occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

#### Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to free this block of memory can result in a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. See if you intended to allocate a memory block to the pointer between the first deallocation and the second. Otherwise, remove the second `free` statement.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before freeing pointers, check them for NULL values and handle the error. In this way, you are protected against freeing an already freed block.

#### Example — Deallocation of Previously Deallocated Pointer Error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

**Correction — Remove Duplicate Deallocation**

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
 }
```

**Example — Freeing Pointer Previously Reallocated With Possibly Zero Size**

```
#include <stdlib.h>

void reshape(char *buf, size_t size) {
  char *reshaped_buf = (char *)realloc(buf, size);
  if (reshaped_buf == NULL) {
    free(buf);
  }
}
```

In this example, the argument `size` of the `reshape()` function can be zero and result in a zero-size reallocation with `realloc()`. In some implementations such as the GNU library, zero-size reallocations free the memory leading to a double free defect.

**Correction – Guard Against Zero-Size Reallocations**

One possible correction is to check size argument of `realloc()` for zero values before use. If the size argument is zero, you can simply free the memory instead of reallocating it.

```
#include <stdlib.h>

void reshape(char *buf, size_t size) {
  if (size != 0) {
    char *reshaped_buf = (char *)realloc(buf, size);
    if (reshaped_buf == NULL) {
      free(buf);
    }
  }
  else {
    free(buf);
  }

}
```

## Check Information

**Category:** Others

## Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-415

# CWE Rule 416

Use After Free

## Description

### Rule Description

*Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code.*

### Polyspace Implementation

The rule checker checks for **Use of previously freed pointer**.

## Examples

### Use of previously freed pointer

#### Issue

This issue occurs when you access a block of memory after freeing the block using the `free` function.

#### Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

#### Example — Use of Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift;
    /* Defect: Reading a freed pointer */
    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

#### Correction — Free Pointer After Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

### Check Information

**Category:** Others

## Version History

**Introduced in R2023a**

### See Also

`Check CWE (-cwe)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

CWE-416

# CWE Rule 457

Use of Uninitialized Variable

## Description

### Rule Description

*The code uses a variable that has not been initialized, leading to unpredictable or unintended results.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Member not initialized in constructor**
- **Non-initialized pointer**
- **Non-initialized variable**

## Examples

### Member not initialized in constructor

#### Issue

This issue occurs when a class constructor has at least one execution path on which it does not initialize some data members of the class.

The defect does not appear in the following cases:

- Empty constructors.
- The non-initialized member is not used in the code.

#### Risk

The members that the constructor does not initialize can have unintended values when you read them later.

Initializing all members in the constructor makes it easier to use your class. If you call a separate method to initialize your members and then read them, you can avoid uninitialized values. However, someone else using your class can read a class member *before* calling your initialization method. Because a constructor is called when you create an object of the class, if you initialize all members in the constructor, they cannot have uninitialized values later on.

#### Fix

The best practice is to initialize all members in your constructor, preferably in an initialization list.

#### Example — Non-Initialized Member

```
class MyClass {
public:
    explicit MyClass(int);
private:
    int _i;
    char _c;
};

MyClass::MyClass(int flag) {
    if(flag == 0) {
        _i = 0;
        _c = 'a';
    }
    else {
        _i = 1;
    }
}
```

In this example, if `flag` is not 0, the member `_c` is not initialized.

The defect appears on the closing brace of the constructor. Following are some tips for navigating in the source code:

- On the **Result Details** pane, see which members are not initialized.
- To navigate to the class definition, right-click a member that is initialized in the constructor. Select **Go To Definition**. In the class definition, you can see all the members, including those members that are not initialized in the constructor.

#### Correction — Initialize All Members on All Execution Paths

One possible correction is to initialize all members of the class `MyClass` for all values of `flag`.

```
class MyClass {
public:
    explicit MyClass(int);
private:
    int _i;
    char _c;
};

MyClass::MyClass(int flag) {
    if(flag == 0) {
        _i = 0;
        _c = 'a';
    }
    else {
        _i = 1;
        _c = 'b';
    }
}
```

### Non-initialized pointer

#### Issue

This issue occurs when a pointer is not assigned an address before dereference.

#### Risk

Unless a pointer is explicitly assigned an address, it points to an unpredictable location.

#### Fix

The fix depends on the root cause of the defect. For instance, you assigned an address to the pointer but the assignment is unreachable.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface" or "Interpret Bug Finder Results in Polyspace Access Web Interface".

See examples of fixes below. It is a good practice to initialize a pointer to NULL when declaring the pointer.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Extend Checker

If a pointer in your code is non-initialized only for certain system input values, you can see one possible combination of input values causing the defect. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

#### Example — Non-initialized pointer error

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j;
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

#### Correction — Initialize Pointer on Every Execution Path

One possible correction is to assign an address to `pi` when `prev` is not `NULL`.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
       {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
       }
    /* Fix: Initialize pi in branches of if statement  */
    else
        pi = prev;


    *pi = j;

    return pi;
}
```

### Non-initialized variable

#### Issue

This issue occurs when a variable is not initialized before its value is read.

#### Risk

Unless a variable is explicitly initialized, the variable value is unpredictable. You cannot rely on the variable having a specific value.

#### Fix

The fix depends on the root cause of the defect. For instance, you assigned a value to the variable but the assignment is unreachable or you assigned a value to the variable in one of two branches of a conditional statement. Fix the unreachable code or missing assignment.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below. It is a good practice to initialize a variable at declaration.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

You can extend the checker in the following ways:

- Polyspace does not flag passing pointers to noninitialized variables to functions. To detect noninitialized variables that are passed to functions by pointers, extend the checker by using the option `-code-behavior-specification`. See “Extend Checkers for Initialization to Check Function Arguments Passed by Pointers”.
- If a variable in your code is non-initialized only for certain system input values, you can see one possible combination of input values causing the defect. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
    /* Defect: val does not have a value if command is not 2 */
}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

#### Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that the initialization is not bypassed on some execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-457

# CWE Rule 460

Improper Cleanup on Thrown Exception

## Description

### Rule Description

*The product does not clean up its state or incorrectly cleans up its state when an exception is thrown, leading to unexpected state or control flow.*

### Polyspace Implementation

The rule checker checks for **Exception violating class invariant**.

## Examples

### Exception violating class invariant

#### Issue

This issue occurs when a non-`noexcept` member function attempts to raise an exception after modifying any of the fields of the class. For instance, Polyspace flags a `throw()` statement or a `new` statement if they are used after modifying the internal state of a class.

#### Risk

To facilitate recovery from an exception while preserving the invariant of the relevant objects, design programs that have exception safety. The C++ standard allows these levels of exception safety:

- Basic Exception Safety: This level of exception safety requires that after raising an exception, the basic invariants of all objects are maintained in a valid state and no memory is leaked. If an operation has basic exception safety, then you can destroy or assign to an object after the operations, even if an exception has been raised. The operations in the standard library offer at least basic exception safety.
- Strong Exception Safety: This level of exception safety requires that after raising an exception, the state of the program remains as it was before the exception. Operations with strong exception safety either succeed or exit with an exception and have no effects on the program.
- `nothrow`: This level of safety requires that an exception cannot occur in an operation.

Without at least the basic exception safety, exceptions might corrupt the state of a program, leave the program in an invalid state, or create memory leaks. Code that does not provide at least the basic exception safety guarantee is unsafe and defective.

#### Fix

To provide at least the basic exception safety in a function, modify the class invariant only when there can be no exceptions. When performing actions that might raise exceptions, use temporary objects that do not modify the original objects invariant.

#### Example

```
#include <cstdint>
#include <cstring>
```

```
template<typename T=int>
class myArray
{
public:
    myArray(){/*...*/}
    myArray(const myArray& rhs)
    {
        DeepCopyNC(rhs);
    }
    ~myArray()
    {
        delete[] array;
    }
    void DeepCopyNC(const myArray& rhs) // Noncompliant
    {
        if (this != &rhs) {
            delete[] array;
            array = nullptr;
            len = rhs.len;

            if (len > 0) {

                array = new T[len];
                std::memcpy(array, rhs.array, len * sizeof(T));
            }
        }
    }

private:
    T* array;
    std::size_t len;
};
extern     myArray<> c1{};
void foo(){

    myArray<> c2{c1};

}
```

This example shows a generic class template `myArray` that manages a raw array. The copy constructor of this class shows two implementation of deep copy. In the function `DeepCopyNC()`, the memory operations are performed after modifying the `this->array` and `this->len` fields. If the memory operation exits with an exception, the invariant of the original class is violated. Because the class invariant is violated by an exception, the function `DeepCopyNC()` cannot guarantee basic exception safety. Polyspace flags it.

**Correction**

To guarantee basic exception safety, modify the class invariant only when the memory operation succeeds with no exceptions, as shown in `DeepCopy`.

```
#include <cstdint>
#include <cstring>
template<typename T=int>
class myArray
{
public:
    myArray(){/*...*/}
    myArray(const myArray& rhs)
    {
        DeepCopy(rhs);
    }
    ~myArray()
    {
        delete[] array;
    }
    void DeepCopy(const myArray& rhs) // Compliant
    {
        T* eTmp = nullptr;

        if (rhs.len > 0) {
            eTmp = new T[rhs.len];
            std::memcpy(eTmp, rhs.array, rhs.len * sizeof(T));
        }

        delete[] array;
        array = eTmp;
        len = rhs.len;
    }

private:
    T* array;
    std::size_t len;
};
extern     myArray<> c1{};
void foo(){

    myArray<> c2{c1};

}
```

## Check Information

**Category:** Initialization and Cleanup Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-460

# CWE Rule 463

Deletion of Data Structure Sentinel

## Description

### Rule Description

*The accidental deletion of a data-structure sentinel can cause serious programming logic problems.*

### Polyspace Implementation

The rule checker checks for **Missing null in string array**.

## Examples

### Missing null in string array

**Issue**

This issue occurs when a string does not have enough space to terminate with a null character `'\0'`.

This defect applies only for projects in C.

**Risk**

A buffer overflow can occur if you copy a string to an array without assuming the implicit null terminator.

**Fix**

If you initialize a character array with a literal, avoid specifying the array bounds.

```
char three[]  = "THREE";
```

The compiler automatically allocates space for a null terminator. In the preceding example, the compiler allocates sufficient space for five characters and a null terminator.

If the issue occurs after initialization, you might have to increase the size of the array by one to account for the null terminator.

In certain circumstances, you might want to initialize the character array with a sequence of characters instead of a string. In this situation, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example — Array size is too small**

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE";
}
```

The character array `three` has a size of 5 and 5 characters `'T'`, `'H'`, `'R'`, `'E'`, and `'E'`. There is no room for the null character at the end because `three` is only five bytes large.

#### Correction — Increase Array Size

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

#### Correction — Change Initialization Method

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

## Check Information

**Category:** Data Neutralization Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-463

# CWE Rule 466

Return of Pointer Value Outside of Expected Range

## Description

### Rule Description

*A function can return a pointer to memory that is outside of the buffer that the pointer is expected to reference.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access out of bounds**
- **Pointer access out of bounds**

## Examples

### Array access out of bounds

#### Issue

This issue occurs when an array index falls outside the range `[0...array_size-1]` during array access.

#### Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

### Example — Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

### Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];

   for (i = 0; i < 10; i++)
    {
        if (i < 2)
            fib[i] = 1;
        else
            fib[i] = fib[i-1] + fib[i-2];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

### Pointer access out of bounds

#### Issue

This issue occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

#### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example — Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

#### Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

## Check Information

**Category:** Pointer Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-466

# CWE Rule 467

Use of sizeof() on a Pointer Type

## Description

### Rule Description

*The code calls sizeof() on a malloced pointer type, which always returns the wordsize/8. This can produce an unexpected result if the programmer intended to determine how much memory has been allocated.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Possible misuse of sizeof**
- **Wrong type used in sizeof**

## Examples

### Possible misuse of sizeof

#### Issue

This issue occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the `sizeof` operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The `sizeof` operator returns the size of that pointer.
- You use the `sizeof` operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as `strncmp` or `wcsncpy` is incorrect because you used the `sizeof` operator earlier with possibly incorrect expectations. For instance:
  - In a function call `strncmp(string1, string2, num)`, `num` is obtained from an incorrect use of the `sizeof` operator on a pointer.
  - In a function call `wcsncpy(destination, source, num)`, `num` is the not the number of wide characters but a size in bytes obtained by using the `sizeof` operator. For instance, you use `wcsncpy(destination, source, sizeof(destination) - 1)` instead of `wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1)`.

#### Risk

Incorrect use of the `sizeof` operator can cause the following issues:

- If you expect the `sizeof` operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of `sizeof` operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.

- If you use the return value of `sizeof` operator incorrectly in a function call, the function does not behave as you expect.

#### Fix

Possible fixes are:

- Do not use the `sizeof` operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the `sizeof` operator carefully to determine the number argument of functions such as `strncmp` or `wcsncpy`. For instance, for wide string functions such as `wcsncpy`, use the number of wide characters as argument instead of the number of bytes.

#### Example — sizeof Used Incorrectly to Determine Array Size

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
    }
}
```

In this example, `sizeof(a)` returns the size of the pointer `a` and not the array size.

#### Correction — Determine Array Size in Another Way

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

### Wrong type used in sizeof

#### Issue

This issue occurs when both of the following conditions hold:

1. You assign the address of a block of memory to a pointer, or transfer data between two blocks of memory. The assignment or copy uses the `sizeof` operator.

   For instance, you initialize a pointer using `malloc(sizeof(`*`type`*`))` or copy data between two addresses using `memcpy(`*`destination_ptr`*`, `*`source_ptr`*`, sizeof(`*`type`*`))`.
2. You use an incorrect type as argument of the `sizeof` operator. For instance:

- You might be using the pointer type instead of the type that the pointer points to. For example, to initialize a *type** pointer, you might be using `malloc(sizeof(`*type*`*))` instead of `malloc(sizeof(`*type*`))`.
- You might be using a completely unrelated type as `sizeof` argument. For example, to initialize a *type** pointer, you might be using `malloc(sizeof(`*anotherType*`))`.

**Risk**

Irrespective of what *type* stands for, the expression `sizeof(`*type*`*)` always returns a fixed size. The size returned is the pointer size on your platform in bytes. The appearance of `sizeof(`*type*`*)` often indicates an unintended usage. The error can cause allocation of a memory block that is much smaller than what you need and lead to weaknesses such as buffer overflows.

For instance, assume that `structType` is a structure with ten `int` variables. If you initialize a `structType*` pointer using `malloc(sizeof(structType*))` on a 32-bit platform, the pointer is assigned a memory block of four bytes. However, to be allocated completely for one `structType` variable, the `structType*` pointer must point to a memory block of `sizeof(structType) = 10 * sizeof(int)` bytes. The required size is much greater than the actual allocated size of four bytes.

**Fix**

To initialize a *type** pointer, replace `sizeof(`*type*`*)` in your pointer initialization expression with `sizeof(`*type*`)`.

**Example — Allocate a Char Array With sizeof**

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char*) * 5);
    free(str);

}
```

In this example, memory is allocated for the character pointer `str` using a `malloc` of five char pointers. However, `str` is a pointer to a character, not a pointer to a character pointer. Therefore the `sizeof` argument, `char*`, is incorrect.

**Correction — Match Pointer Type to `sizeof` Argument**

One possible correction is to match the argument to the pointer type. In this example, `str` is a character pointer, therefore the argument must also be a character.

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char) * 5);
    free(str);

}
```

## Check Information

**Category:** Pointer Issues

## Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-467

# CWE Rule 468

Incorrect Pointer Scaling

## Description

### Rule Description

*In C and C++, one may often accidentally refer to the wrong memory due to the semantics of when math operations are implicitly scaled.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Array access out of bounds**
- **Incorrect pointer scaling**
- **Pointer access out of bounds**
- **Subtraction or comparison between pointers to different arrays**

## Examples

### Array access out of bounds

#### Issue

This issue occurs when an array index falls outside the range `[0...array_size-1]` during array access.

#### Risk

Accessing an array outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you accessed an array inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used an array index that is the same as the loop index instead of being one less than the loop index.

To fix the issue, you have to modify the loop bound or the array index.

Another reason why an array index can exceed array bounds is a prior conversion from signed to unsigned integers. The conversion can result in a wrap around of the index value, eventually causing the array index to exceed the array bounds.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Array Access Out of Bounds Error

```
#include <stdio.h>

void fibonacci(void)
{
    int i;
    int fib[10];

    for (i = 0; i < 10; i++)
       {
        if (i < 2)
            fib[i] = 1;
         else
            fib[i] = fib[i-1] + fib[i-2];
       }

    printf("The 10-th Fibonacci number is %i .\n", fib[i]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

The array `fib` is assigned a size of 10. An array index for `fib` has allowed values of `[0,1,2,...,9]`. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, the `printf` statement attempts to access `fib[10]` through `i`.

#### Correction — Keep Array Index Within Array Bounds

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
   int i;
   int fib[10];
  for (i = 0; i < 10; i++)
   {
       if (i < 2)
           fib[i] = 1;
       else
           fib[i] = fib[i-1] + fib[i-2];
   }

   /* Fix: Print fib[9] instead of fib[10] */
   printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

### Incorrect pointer scaling

#### Issue

This issue occurs when Polyspace Bug Finder considers that you are ignoring the implicit scaling in pointer arithmetic.

For instance, the defect can occur in the following situations.

| Situation | Risk | Possible Fix |
|---|---|---|
| You use the `sizeof` operator in arithmetic operations on a pointer. | The `sizeof` operator returns the size of a data type in number of bytes.<br>Pointer arithmetic is already implicitly scaled by the size of the data type of the pointed variable. Therefore, the use of `sizeof` in pointer arithmetic produces unintended results. | Do not use `sizeof` operator in pointer arithmetic. |
| You perform arithmetic operations on a pointer, and then apply a cast. | Pointer arithmetic is implicitly scaled. If you do not consider this implicit scaling, casting the result of a pointer arithmetic produces unintended results. | Apply the cast before the pointer arithmetic. |

#### Fix

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Example — Use of sizeof Operator

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2*(sizeof(int)));
}
```

In this example, the operation `2*(sizeof(int))` returns twice the size of an `int` variable in bytes. However, because pointer arithmetic is implicitly scaled, the number of bytes by which `ptr` is offset is `2*(sizeof(int))*(sizeof(int))`.

In this example, the incorrect scaling shifts `ptr` outside the bounds of the array. Therefore, a **Pointer access out of bounds** error appears on the * operation.

### Correction — Remove `sizeof` Operator

One possible correction is to remove the `sizeof` operator.

```
void func(void) {
    int arr[5] = {1,2,3,4,5};
    int *ptr = arr;

    int value_in_position_2 = *(ptr + 2);
}
```

### Example — Cast Following Pointer Arithmetic

```
int func(void) {
    int x = 0;
    char r = *(char *)(&x + 1);
    return r;
}
```

In this example, the operation `&x + 1` offsets `&x` by `sizeof(int)`. Following the operation, the resulting pointer points outside the allowed buffer. When you dereference the pointer, a **Pointer access out of bounds** error appears on the * operation.

### Correction — Apply Cast Before Pointer Arithmetic

If you want to access the second byte of `x`, first cast `&x` to a `char*` pointer and then perform the pointer arithmetic. The resulting pointer is offset by `sizeof(char)` bytes and still points within the allowed buffer, whose size is `sizeof(int)` bytes.

```
int func(void) {
    int x = 0;
    char r = *((char *)(&x )+ 1);
    return r;
}
```

## Pointer access out of bounds

### Issue

This issue occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

#### Risk

Dereferencing a pointer outside its bounds is undefined behavior. You can read an unpredictable value or try to access a location that is not allowed and encounter a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. For instance, you dereferenced a pointer inside a loop and one of these situations happened:

- The upper bound of the loop is too large.
- You used pointer arithmetic to advance the pointer with an incorrect value for the pointer increment.

To fix the issue, you have to modify the loop bound or the pointer increment value.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface" or "Interpret Bug Finder Results in Polyspace Access Web Interface".

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

#### Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

### Subtraction or comparison between pointers to different arrays

#### Issue

This issue occurs when you subtract or compare pointers that are null or that point to elements in different arrays. The relational operators for the comparison are >, <, >=, and <=.

#### Risk

When you subtract two pointers to elements in the same array, the result is the difference between the subscripts of the two array elements. Similarly, when you compare two pointers to array elements, the result is the positions of the pointers relative to each other. If the pointers are null or point to different arrays, a subtraction or comparison operation is undefined. If you use the subtraction result as a buffer index, it can cause a buffer overflow.

#### Fix

Before you subtract or use relational operators to compare pointers to array elements, check that they are non-null and that they point to the same array.

#### Example — Subtraction Between Pointers to Elements in Different Arrays

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int end;
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation is undefined unless array nums
    is adjacent to variable end in memory. */
    free_elements = &end - next_num_ptr;
    return free_elements;
}
```

In this example, the array nums is incrementally filled. Pointer subtraction is then used to determine how many free elements remain. Unless end points to a memory location one past the last element of nums, the subtraction operation is undefined.

#### Correction — Subtract Pointers to the Same Array

Subtract the pointer to the last element that was filled from the pointer to the last element in the array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation involves pointers to the same array. */
    free_elements = &(nums[SIZE20 - 1]) - next_num_ptr;

    return free_elements + 1;
}
```

## Check Information

**Category:** Pointer Issues

# Version History

**Introduced in R2023a**

## See Also

Check CWE (-cwe)

#### Topics

“Check for and Review Coding Standard Violations”

#### External Websites

CWE-468

# CWE Rule 469

Use of Pointer Subtraction to Determine Size

## Description

### Rule Description

*The application subtracts one pointer from another in order to determine size, but this calculation can be incorrect if the pointers do not exist in the same memory chunk.*

### Polyspace Implementation

The rule checker checks for **Subtraction or comparison between pointers to different arrays**.

## Examples

### Subtraction or comparison between pointers to different arrays

**Issue**

This issue occurs when you subtract or compare pointers that are null or that point to elements in different arrays. The relational operators for the comparison are >, <, >=, and <=.

**Risk**

When you subtract two pointers to elements in the same array, the result is the difference between the subscripts of the two array elements. Similarly, when you compare two pointers to array elements, the result is the positions of the pointers relative to each other. If the pointers are null or point to different arrays, a subtraction or comparison operation is undefined. If you use the subtraction result as a buffer index, it can cause a buffer overflow.

**Fix**

Before you subtract or use relational operators to compare pointers to array elements, check that they are non-null and that they point to the same array.

**Example — Subtraction Between Pointers to Elements in Different Arrays**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int end;
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation is undefined unless array nums
    is adjacent to variable end in memory. */
    free_elements = &end - next_num_ptr;
    return free_elements;
}
```

In this example, the array `nums` is incrementally filled. Pointer subtraction is then used to determine how many free elements remain. Unless `end` points to a memory location one past the last element of `nums`, the subtraction operation is undefined.

#### Correction — Subtract Pointers to the Same Array

Subtract the pointer to the last element that was filled from the pointer to the last element in the array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE20 20

size_t func(void)
{
    int nums[SIZE20];
    int *next_num_ptr = nums;
    size_t free_elements;
    /* Increment next_num_ptr as array fills */

    /* Subtraction operation involves pointers to the same array. */
    free_elements = &(nums[SIZE20 - 1]) - next_num_ptr;

    return free_elements + 1;
}
```

## Check Information

**Category:** Pointer Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-469

# CWE Rule 474

Use of Function with Inconsistent Implementations

## Description

### Rule Description

*The code uses a function that has inconsistent implementations across operating systems and versions.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Signal call from within signal handler**
- **Use of obsolete standard function**

## Examples

### Signal call from within signal handler

**Issue**

This issue occurs when you call the function `signal()` from a signal handler on Windows platforms.

The defect is detected only if you specify a Visual Studio compiler. See `Compiler (-compiler)`.

**Risk**

The function `signal()` associates a signal with a signal handler function. On platforms such as Windows, which removes this association after receiving the signal, you might call the function `signal()` again *within the signal handler* to re-establish the association.

However, this attempt to make a signal handler persistent is prone to race conditions. On Windows platforms, from the time the signal handler begins execution to when the `signal` function is called again, it is the default signal handling, `SIG_DFL`, that is active. If a second signal is received within this time window, you see the default signal handling and not the custom signal handler, but you might expect otherwise.

**Fix**

Do not call `signal()` from a signal handler on Windows platforms.

**Example — signal() Called from Signal Handler**

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;
```

```
void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;

    /* Call signal() to reestablish sig_handler
    upon receiving SIG_ERR. */

    if (signal(s0, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
}

void func(void)
{
        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
  /* more code */
}
```

In this example, the definition of `sig_handler()` includes a call to `signal()` when the handler catches `SIG_ERR`. On Windows platforms, signal handlers are nonpersistent. This code can result in a race condition.

The issue is detected only if you specify a compiler such as `visual15.x` for the analysis.

#### Correction — Do Not Call `signal()` from Signal Handler

Avoid attempting to make a signal handler persistent on Windows. If your code requires the use of a persistent signal handler on a Windows platform, use a persistent signal handler after performing a thorough risk analysis.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>



volatile sig_atomic_t e_flag = 0;


void sig_handler(int signum)
{
    int s0 = signum;
    e_flag = 1;
    /* No call to signal() */
}

int main(void)
{
        if (signal(SIGINT, sig_handler) == SIG_ERR)
        {
            /* Handle error */

        }
}
```

### Use of obsolete standard function

#### Issue

This issue occurs when you use standard function routines that are considered legacy, removed, deprecated, or obsolete by C/C++ coding standards.

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| `asctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |
| `asctime_r` | Deprecated in POSIX.1-2008 | Implementation based on unsafe function `sprintf`. | `strftime` or `asctime_s` |
| `bcmp` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcmp` |
| `bcopy` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcpy` or `memmove` |
| `brk` and `sbrk` | Marked as legacy in SUSv2 and POSIX.1-2001. | | `malloc` |
| `bsd_signal` | Removed in POSIX.1-2008 | | `sigaction` |
| `bzero` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | `memset` |
| `ctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |
| `ctime_r` | Deprecated in POSIX.1-2008 | Implementation based on unsafe function `sprintf`. | `strftime` or `asctime_s` |
| `cuserid` | Removed in POSIX.1-2001. | Not reentrant. Precise functionality not standardized causing portability issues. | `getpwuid` |
| `ecvt` and `fcvt` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | Not reentrant | `snprintf` |
| `ecvt_r` and `fcvt_r` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | | `snprintf` |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| ftime | Removed in POSIX.1-2008 | | time, gettimeofday, clock_gettime |
| gamma, gammaf, gammal | Function not specified in any standard because of historical variations | Portability issues. | tgamma, lgamma |
| gcvt | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | snprintf |
| getcontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| getdtablesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_OPEN _MAX ) |
| gethostbyaddr | Removed in POSIX.1-2008 | Not reentrant | getaddrinfo |
| gethostbyname | Removed in POSIX.1-2008 | Not reentrant | getnameinfo |
| getpagesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_PAGE SIZE ) |
| getpass | Removed in POSIX.1-2001. | Not reentrant. | getpwuid |
| getw | Not present in POSIX.1-2001. | | fread |
| getwd | Marked legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | getcwd |
| index | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strchr |
| makecontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| memalign | Appears in SunOS 4.1.3. Not in 4.4 BSD or POSIX.1-2001 | | posix_memalign |
| mktemp | Removed in POSIX.1-2008. | Generated names are predictable and can cause a race condition. | mkstemp removes race risk |
| pthread_attr_ getstackaddr and pthread_attr_ setstackaddr | | Ambiguities in the specification of the stackaddr attribute cause portability issues | pthread_attr_ getstack and pthread_attr_ setstack |
| putw | Not present in POSIX.1-2001. | Portability issues. | fwrite |
| qecvt and qfcvt | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | snprintf |
| qecvt_r and qfcvt_r | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | snprintf |
| rand_r | Marked as obsolete in POSIX.1-2008 | | |
| re_comp | BSD API function | Portability issues | regcomp |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| re_exes | BSD API function | Portability issues | regexec |
| rindex | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strrchr |
| scalb | Removed in POSIX.1-2008 | | scalbln, scalblnf, or scalblnl |
| sigblock | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigmask | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigsetmask | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigstack | Interface is obsolete and not implemented on most platforms. | Portability issues. | sigaltstack |
| sigvec | 4.3BSD signal API whose origin is unclear | | sigaction |
| swapcontext | Removed in POSIX.1-2008 | Portability issues. | Use POSIX threads. |
| tmpnam and tmpnam_r | Marked as obsolete in POSIX.1-2008. | This function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined. | mkstemp, tmpfile |
| ttyslot | Removed in POSIX.1-2001. | | |
| ualarm | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | Errors are under-specified | setitimer or POSIX timer_create |
| usleep | Removed in POSIX.1-2008. | | nanosleep |
| utime | SVr4, POSIX.1-2001. POSIX.1-2008 marks as obsolete. | | |
| valloc | Marked as obsolete in 4.3BSD.<br>Marked as legacy in SUSv2.<br>Removed from POSIX.1-2001 | | posix_memalign |
| vfork | Removed from POSIX.1-2008 | Under-specified in previous standards. | fork |
| wcswcs | This function was not included in the final ISO/IEC 9899:1990/ Amendment 1:1995 (E). | | wcsstr |
| WinExec | WinAPI provides this function only for 16-bit Windows compatibility. | | CreateProcess |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| `LoadModule` | WinAPI provides this function only for 16-bit Windows compatibility. | | `CreateProcess` |

#### Fix

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Printing Out Time

```
#include <stdio.h>
#include <time.h>

void timecheck_bad(int argc, char *argv[])
{
    time_t ticks;

    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks));
}
```

In this example, the function `ctime` formats the current time and prints it out. However, `ctime` was removed after C99 because it does not work on multithreaded programs.

#### Correction — Different Time Function

One possible correction is to use `strftime` instead because this function uses a set buffer size.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

void timecheck_good(int argc, char *argv[])
{
    char outBuff[1025];
    time_t ticks;
    struct tm * timeinfo;

    memset(outBuff, 0, sizeof(outBuff));

    ticks = time(NULL);
    timeinfo = localtime(&ticks);
    strftime(outBuff,sizeof(outBuff),"%I:%M%p.",timeinfo);
    fprintf(stdout, outBuff);
}
```

## Check Information

**Category:** API / Function Errors

## Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-474

# CWE Rule 476

NULL Pointer Dereference

## Description

### Rule Description

*A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.*

### Polyspace Implementation

The rule checker checks for **Dereference of a null pointer**.

## Examples

### Dereference of a null pointer

#### Issue

This issue occurs when you use a pointer with a value of `NULL` as if it points to a valid memory location. If you dereference the zero address, such as 0x00, Polyspace considers the null address as equivalent to `NULL` and raises this defect.

#### Risk

Dereferencing a null pointer is undefined behavior. In most implementations, the dereference can cause your program to crash.

#### Fix

Check a pointer for `NULL` before dereference.

If the issue occurs despite an earlier check for `NULL`, look for intermediate events between the check and the subsequent dereference. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Null pointer error

```
#include <stdlib.h>
```

```
int FindMax(int *arr, int Size)
{
 int* p=NULL;

 *p=arr[0];
 /* Defect: Null pointer dereference */

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

The pointer p is initialized with value of NULL. However, when the value arr[0] is written to *p, p is assumed to point to a valid memory location.

#### Correction — Assign Address to Null Pointer Before Dereference

One possible correction is to initialize p with a valid memory address before dereference.

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 /* Fix: Assign address to null pointer */
 int* p=&arr[0];

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

## Check Information

**Category:** Pointer Issues

# Version History

**Introduced in R2023a**

## See Also

Check CWE (-cwe)

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-476

# CWE Rule 477

Use of Obsolete Function

## Description

### Rule Description

*The code uses deprecated or obsolete functions, which suggests that the code has not been actively reviewed or maintained.*

### Polyspace Implementation

The rule checker checks for **Use of obsolete standard function**.

## Examples

### Use of obsolete standard function

#### Issue

This issue occurs when you use standard function routines that are considered legacy, removed, deprecated, or obsolete by C/C++ coding standards.

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| `asctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |
| `asctime_r` | Deprecated in POSIX.1-2008 | Implementation based on unsafe function `sprintf`. | `strftime` or `asctime_s` |
| `bcmp` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcmp` |
| `bcopy` | Deprecated in 4.3BSD<br>Marked as legacy in POSIX.1-2001. | Returns from function after finding the first differing byte, making it vulnerable to timing attacks. | `memcpy` or `memmove` |
| `brk` and `sbrk` | Marked as legacy in SUSv2 and POSIX.1-2001. | | `malloc` |
| `bsd_signal` | Removed in POSIX.1-2008 | | `sigaction` |
| `bzero` | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | `memset` |
| `ctime` | Deprecated in POSIX.1-2008 | Not thread-safe. | `strftime` or `asctime_s` |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| ctime_r | Deprecated in POSIX.1-2008 | Implementation based on unsafe function sprintf. | strftime or asctime_s |
| cuserid | Removed in POSIX.1-2001. | Not reentrant. Precise functionality not standardized causing portability issues. | getpwuid |
| ecvt and fcvt | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | Not reentrant | snprintf |
| ecvt_r and fcvt_r | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008 | | snprintf |
| ftime | Removed in POSIX.1-2008 | | time, gettimeofday, clock_gettime |
| gamma, gammaf, gammal | Function not specified in any standard because of historical variations | Portability issues. | tgamma, lgamma |
| gcvt | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | snprintf |
| getcontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| getdtablesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_OPEN_MAX ) |
| gethostbyaddr | Removed in POSIX.1-2008 | Not reentrant | getaddrinfo |
| gethostbyname | Removed in POSIX.1-2008 | Not reentrant | getnameinfo |
| getpagesize | BSD API function not included in POSIX.1-2001 | Portability issues. | sysconf( _SC_PAGESIZE ) |
| getpass | Removed in POSIX.1-2001. | Not reentrant. | getpwuid |
| getw | Not present in POSIX.1-2001. | | fread |
| getwd | Marked legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | getcwd |
| index | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strchr |
| makecontext | Removed in POSIX.1-2008. | Portability issues. | Use POSIX thread instead. |
| memalign | Appears in SunOS 4.1.3. Not in 4.4 BSD or POSIX.1-2001 | | posix_memalign |
| mktemp | Removed in POSIX.1-2008. | Generated names are predictable and can cause a race condition. | mkstemp removes race risk |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| pthread_attr_ getstackaddr and pthread_attr_ setstackaddr | | Ambiguities in the specification of the stackaddr attribute cause portability issues | pthread_attr_ getstack and pthread_attr_ setstack |
| putw | Not present in POSIX.1-2001. | Portability issues. | fwrite |
| qecvt and qfcvt | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | snprintf |
| qecvt_r and qfcvt_r | Marked as legacy in POSIX.1-2001, removed in POSIX.1-2008 | | snprintf |
| rand_r | Marked as obsolete in POSIX.1-2008 | | |
| re_comp | BSD API function | Portability issues | regcomp |
| re_exes | BSD API function | Portability issues | regexec |
| rindex | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | | strrchr |
| scalb | Removed in POSIX.1-2008 | | scalbln, scalblnf, or scalblnl |
| sigblock | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigmask | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigsetmask | 4.3BSD signal API whose origin is unclear | | sigprocmask |
| sigstack | Interface is obsolete and not implemented on most platforms. | Portability issues. | sigaltstack |
| sigvec | 4.3BSD signal API whose origin is unclear | | sigaction |
| swapcontext | Removed in POSIX.1-2008 | Portability issues. | Use POSIX threads. |
| tmpnam and tmpnam_r | Marked as obsolete in POSIX.1-2008. | This function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined. | mkstemp, tmpfile |
| ttyslot | Removed in POSIX.1-2001. | | |
| ualarm | Marked as legacy in POSIX.1-2001. Removed in POSIX.1-2008. | Errors are under-specified | setitimer or POSIX timer_create |
| usleep | Removed in POSIX.1-2008. | | nanosleep |
| utime | SVr4, POSIX.1-2001. POSIX.1-2008 marks as obsolete. | | |

| Obsolete Function | Standards | Risk | Replacement Function |
|---|---|---|---|
| `valloc` | Marked as obsolete in 4.3BSD.<br>Marked as legacy in SUSv2.<br>Removed from POSIX.1-2001 | | `posix_memalign` |
| `vfork` | Removed from POSIX.1-2008 | Under-specified in previous standards. | `fork` |
| `wcswcs` | This function was not included in the final ISO/IEC 9899:1990/ Amendment 1:1995 (E). | | `wcsstr` |
| `WinExec` | WinAPI provides this function only for 16-bit Windows compatibility. | | `CreateProcess` |
| `LoadModule` | WinAPI provides this function only for 16-bit Windows compatibility. | | `CreateProcess` |

**Fix**

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example — Printing Out Time**

```
#include <stdio.h>
#include <time.h>

void timecheck_bad(int argc, char *argv[])
{
    time_t ticks;

    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks));
}
```

In this example, the function `ctime` formats the current time and prints it out. However, `ctime` was removed after C99 because it does not work on multithreaded programs.

**Correction — Different Time Function**

One possible correction is to use `strftime` instead because this function uses a set buffer size.

```
#include <stdio.h>
#include <string.h>
#include <time.h>
```

```
void timecheck_good(int argc, char *argv[])
{
    char outBuff[1025];
    time_t ticks;
    struct tm * timeinfo;

    memset(outBuff, 0, sizeof(outBuff));

    ticks = time(NULL);
    timeinfo = localtime(&ticks);
    strftime(outBuff,sizeof(outBuff),"%I:%M%p.",timeinfo);
    fprintf(stdout, outBuff);
}
```

## Check Information

**Category:** API / Function Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-477

# CWE Rule 478

Missing Default Case in Multiple Condition Expression

## Description

### Rule Description

*The code does not have a default case in an expression with multiple conditions, such as a switch statement.*

### Polyspace Implementation

The rule checker checks for **Missing case for switch condition**.

## Examples

### Missing case for switch condition

#### Issue

This issue occurs when the `switch` variable can take values that are not covered by a `case` statement.

**Note** Bug Finder only raises a defect if the switch variable is not full range.

#### Risk

If the `switch` variable takes a value that is not covered by a `case` statement, your program can have unintended behavior.

A switch-statement that makes a security decision is particularly vulnerable when all possible values are not explicitly handled. An attacker can use this situation to deviate the normal execution flow.

#### Fix

It is good practice to use a `default` statement as a catch-all for values that are not covered by a `case` statement. Even if the `switch` variable takes an unintended value, the resulting behavior can be anticipated.

#### Example — Missing Default Condition

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;
```

```
static LOGIN system_access(const char *username) {
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}

int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    }

    printf("Welcome!\n");
    return r;
}
```

In this example, the `enum` parameter `User` can take a value `UNKNOWN` that is not covered by a `case` statement.

**Correction — Add a Default Condition**

One possible correction is to add a default condition for possible values that are not covered by a `case` statement.

```
#include <stdio.h>
#include <string.h>

typedef enum E
{
    ADMIN=1,
    GUEST,
    UNKNOWN = 0
} LOGIN;

static LOGIN system_access(const char *username) {
  LOGIN user = UNKNOWN;

  if ( strcmp(username, "root") == 0 )
    user = ADMIN;

  if ( strcmp(username, "friend") == 0 )
    user = GUEST;

  return user;
}
```

```
int identify_bad_user(const char * username)
{
    int r=0;

    switch( system_access(username) )
    {
    case ADMIN:
        r = 1;
        break;
    case GUEST:
        r = 2;
    break;
    default:
        printf("Invalid login credentials!\n");
    }

    printf("Welcome!\n");
    return r;
}
```

## Check Information

**Category:** Bad Coding Practices

# Version History

**Introduced in R2023a**

## See Also

Check CWE (-cwe)

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-478

# CWE Rule 479

Signal Handler Use of a Non-reentrant Function

## Description

### Rule Description

*The program defines a signal handler that calls a non-reentrant function.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Function called from signal handler not asynchronous-safe**
- **Function called from signal handler not asynchronous-safe (strict ISO C)**

## Examples

### Function called from signal handler not asynchronous-safe

**Issue**

This issue occurs when a signal handler calls a function that is not asynchronous-safe according to the POSIX standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

**Risk**

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

**Fix**

The POSIX standard defines these functions as asynchronous-safe. You can call these functions from a signal handler.

| | | |
|---|---|---|
| `_exit()` | `getpgrp()` | `setsockopt()` |
| `_Exit()` | `getpid()` | `setuid()` |
| `abort()` | `getppid()` | `shutdown()` |
| `accept()` | `getsockname()` | `sigaction()` |
| `access()` | `getsockopt()` | `sigaddset()` |
| `aio_error()` | `getuid()` | `sigdelset()` |

| | | |
|---|---|---|
| aio_return() | kill() | sigemptyset() |
| aio_suspend() | link() | sigfillset() |
| alarm() | linkat() | sigismember() |
| bind() | listen() | signal() |
| cfgetispeed() | lseek() | sigpause() |
| cfgetospeed() | lstat() | sigpending() |
| cfsetispeed() | mkdir() | sigprocmask() |
| cfsetospeed() | mkdirat() | sigqueue() |
| chdir() | mkfifo() | sigset() |
| chmod() | mkfifoat() | sigsuspend() |
| chown() | mknod() | sleep() |
| clock_gettime() | mknodat() | sockatmark() |
| close() | open() | socket() |
| connect() | openat() | socketpair() |
| creat() | pathconf() | stat() |
| dup() | pause() | symlink() |
| dup2() | pipe() | symlinkat() |
| execl() | poll() | sysconf() |
| execle() | posix_trace_event() | tcdrain() |
| execv() | pselect() | tcflow() |
| execve() | pthread_kill() | tcflush() |
| faccessat() | pthread_self() | tcgetattr() |
| fchdir() | pthread_sigmask() | tcgetpgrp() |
| fchmod() | quick_exit() | tcsendbreak() |
| fchmodat() | raise() | tcsetattr() |
| fchown() | read() | tcsetpgrp() |
| fchownat() | readlink() | time() |
| fcntl() | readlinkat() | timer_getoverrun() |
| fdatasync() | recv() | timer_gettime() |
| fexecve() | recvfrom() | timer_settime() |
| fork() | recvmsg() | times() |
| fpathconf() | rename() | umask() |
| fstat() | renameat() | uname() |
| fstatat() | rmdir() | unlink() |
| fsync() | select() | unlinkat() |
| ftruncate() | sem_post() | utime() |
| futimens() | send() | utimensat() |

| | | |
|---|---|---|
| getegid() | sendmsg() | utimes() |
| geteuid() | sendto() | wait() |
| getgid() | setgid() | waitpid() |
| getgroups() | setpgid() | write() |
| getpeername() | setsid() | |

Functions not in the previous table are not asynchronous-safe, and should not be called from a signal hander.

**Example — Call to printf() Inside Signal Handler**

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler(int signum)
{
    /* Call function printf() that is not
    asynchronous-safe */
    printf("signal %d received.", signum);
    e_flag = 1;
}

int main(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, sizeof(char));
    if (info == NULL)
    {
        /* Handle Error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

In this example, `sig_handler` calls `printf()` when catching a signal. If the handler catches another signal while `printf()` is executing, the behavior of the program is undefined.

**Correction — Set Flag Only in Signal Handler**

Use your signal handler to set only the value of a flag. `e_flag` is of type volatile `sig_atomic_t`. `sig_handler` can safely access it asynchronously.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

#define SIZE20 20

extern volatile sig_atomic_t e_flag;

void display_info(const char *info)
{
    if (info)
    {
        (void)fputs(info, stderr);
    }
}

void sig_handler1(int signum)
{
    int s0 = signum;
    e_flag = 1;
}

int func(void)
{
    e_flag = 0;
    if (signal(SIGINT, sig_handler1) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info = (char *)calloc(SIZE20, 1);
    if (info == NULL)
    {
        /* Handle error */
    }
    while (!e_flag)
    {
        /* Main loop program code */
        display_info(info);
        /* More program code */
    }
    free(info);
    info = NULL;
    return 0;
}
```

### Function called from signal handler not asynchronous-safe (strict ISO C)

#### Issue

This issue occurs when a signal handler calls a function that is not asynchronous-safe according to the C standard. An asynchronous-safe function can be interrupted at any point in its execution, then called again without causing an inconsistent state. It can also correctly handle global data that might be in an inconsistent state.

The C standard defines a stricter subset of functions as asynchronous-safe compared to the set of functions that are asynchronous-safe according to the POSIX standard. **Function called from signal handler not asynchronous-safe (strict ISO C)** reports a defect when a signal handler calls any function that is not part of that subset, even if the function is asynchronous-safe according to the POSIX standard.

To check for calls to functions that are not asynchronous-safe according to the POSIX standard, enable checker **Function called from signal handler not asynchronous-safe**.

If a signal handler calls another function that calls an asynchronous-unsafe function, the defect appears on the function call in the signal handler. The defect traceback shows the full path from the signal handler to the asynchronous-unsafe function.

#### Risk

When a signal handler is invoked, the execution of the program is interrupted. After the handler is finished, program execution resumes at the point of interruption. If a function is executing at the time of the interruption, calling it from within the signal handler is undefined behavior, unless it is asynchronous-safe.

#### Fix

The C standard defines the following functions as asynchronous-safe. You can call these functions from a signal handler:

- `abort()`
- `_Exit()`
- `quick_exit()`
- `signal()`

#### Example — Call to raise() Inside Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>
```

```
void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}

void sig_handler(int signum)
{
    int s0 = signum;
    /* Call raise() */
    if (raise(SIGTERM) != 0) {
        /* Handle error */
    }
}

int finc(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

In this example, `sig_handler` calls `raise()` when catching a signal. If the handler catches another signal while `raise()` is executing, the behavior of the program is undefined.

**Correction — Remove Call to `raise()` in Signal Handler**

According to the C standard, the only functions that you can safely call from a signal handler are `abort()`, `_Exit()`, `quick_exit()`, and `signal()`.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <setjmp.h>
#include <syslog.h>
#include <unistd.h>

void SIG_ERR_handler(int signum)
{
    int s0 = signum;
    /* SIGTERM specific handling */
}
void sig_handler(int signum)
```

```
{
    int s0 = signum;


}

int func(void)
{
    if (signal(SIGTERM, SIG_ERR_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if (signal(SIGINT, sig_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return 0;
}
```

## Check Information

**Category:** Handler Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-479

# CWE Rule 480

Use of Incorrect Operator

## Description

### Rule Description

*The programmer accidentally uses the wrong operator, which changes the application logic in security-relevant ways.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Invalid use of = (assignment) operator**
- **Invalid use of == (equality) operator**
- **Use of bitwise operator with a Boolean-like operand**

## Examples

### Invalid use of = (assignment) operator

#### Issue

This issue occurs when an assignment is made inside the predicate of a conditional, such as `if` or `while`.

In C and C++, a single equal sign is an assignment not a comparison. Using a single equal sign in a conditional statement can indicate a typo or a mistake.

#### Risk

- Conditional statement tests the wrong values— The single equal sign operation assigns the value of the right operand to the left operand. Then, because this assignment is inside the predicate of a conditional, the program checks whether the new value of the left operand is nonzero or not NULL.
- Maintenance and readability issues — Even if the assignment is intended, someone reading or updating the code can misinterpret the assignment as an equality comparison instead of an assignment.

#### Fix

- If the assignment is a bug, to check for equality, add a second equal sign (==).
- If the assignment inside the conditional statement was intentional, to improve readability, separate the assignment and the test. Move the assignment outside the control statement. In the control statement, simply test the result of the assignment.

  If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Single Equal Sign Inside an if Condition

```
#include <stdio.h>

void bad_equals_ex(int alpha, int beta)
{
    if(alpha = beta)
    {
        printf("Equal\n");
    }
}
```

The equal sign is flagged as a defect because the assignment operator is used within the predicate of the if-statement. The predicate assigns the value `beta` to `alpha`, then implicitly tests whether `alpha` is true or false.

#### Correction — Change Expression to Comparison

One possible correction is adding an additional equal sign. This correction changes the assignment to a comparison. The if condition compares whether `alpha` and `beta` are equal.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha == beta)
    {
        printf("Equal\n");
    }
}
```

#### Correction — Assignment and Comparison Inside the `if` Condition

If an assignment must be made inside the predicate, a possible correction is adding an explicit comparison. This correction assigns the value of `beta` to `alpha`, then explicitly checks whether `alpha` is nonzero. The code is clearer.

```
#include <stdio.h>

int assignment_not_zero(int alpha, int beta)
{
    if((alpha = beta) != 0)
    {
        return alpha;
    }
    else
    {
        return 0;
    }
}
```

#### Correction — Move Assignment Outside the `if` Statement

If the assignment can be made outside the control statement, one possible correction is to separate the assignment and comparison. This correction assigns the value of `beta` to `alpha` before the if. Inside the if-condition, only `alpha` is given to test if `alpha` is nonzero or not NULL.

```
#include <stdio.h>

void assign_and_print(int alpha, int beta)
{
    alpha = beta;
    if(alpha)
    {
        printf("%d", alpha);
    }
}
```

### Invalid use of == (equality) operator

#### Issue

This issue occurs when you use an equality operator instead of an assignment operator in a simple statement.

#### Risk

The use of == operator instead of an = operator can silently produce incorrect results. If you intended to assign a value to a variable, the assignment does not occur. The variable retains its previous value or if not initialized previously, stays uninitialized.

#### Fix

Use the = (assignment) operator instead of the == (equality) operator.

The check appears on chained assignment and equality operators such as:

```
compFlag = val1 == val2;
```

For better readability of your code, place the equality check in parenthesis.

```
compFlag = (val1 == val2);
```

If the use of == operator is intended, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example — Equality Evaluation in for-Loop

```
void populate_array(void)
{
    int i = 0;
    int j = 0;
    int array[4];
    for (j == 5; j < 9; j++) {
        array[i] = j;
        i++;
    }
}
```

Inside the `for`-loop, the statement `j == 5` tests whether `j` is equal to 5 instead of setting `j` to 5. The `for`-loop iterates from 0 to 8 because `j` starts with a value of 0, not 5. A by-product of the invalid equality operator is an out-of-bounds array access in the next line.

#### Correction — Change to Assignment Operator

One possible correction is to change the == operator to a single equal sign (=). Changing the == sign resolves both defects because the `for`-loop iterates the intended number of times.

```
void populate_array(void)
{
    int i = 0;
    int j = 0;
    int array[4];

    for (j = 5; j < 9; j++) {
        array[i] = j;
        i++;
    }
}
```

### Use of bitwise operator with a Boolean-like operand

#### Issue

This issue occurs when you use bitwise operators, such as:

- Bitwise AND (&, &=)
- Bitwise OR (`|`, `|=`)
- Bitwise XOR (^, ^=)
- Bitwise NOT(~)

with:

- Boolean type variables
- Outputs of relational or equality expressions

Using Boolean type variables as array indices, in Boolean arithmetic expression, and in shifting operations does not raise this defect.

#### Risk

Boolean-like operands, such as variables of type `bool` and outputs of relational operators typically appear in logical expressions. Using a bitwise operator in an expression containing Boolean variables and relational operators might be a sign of logic error. Because bitwise operators and logical operators look similar, you might inadvertently use a bitwise operator instead of a logical operator. Such logic errors do not raise any compilation error and can introduce bugs in your code that are difficult to find.

#### Fix

Use logical operators in expressions that contain Boolean variables and relational operator. To indicate that you intend to use a bitwise operator in such an expression, use parentheses.

#### Example — Possible Bug Due to Using Bitwise Operator

```
class User{
    //...
    int uid;
    int euid;
public:
    int getuid();
    int geteuid();
};
void Noncompliant ()
{
    User nU;
    if (nU.getuid () & nU.geteuid () == 0) {   //Noncompliant
        //...
    }else{
        //...
    }
}
```

In this example, the `if-else` block is executed conditionally. The conditional statement uses the bitwise AND (&) instead of the logical AND (&&), perhaps by mistake. Consider when the function `nU.geteuid()` evaluates to 0, and `nU.getuid()` evaluates to 2. In this case, the `else` block of code executes if you use & because `2&1` evaluates to `false`. Conversely, the `if` block of code executes when you use && because `2&&1` evaluates to `true`. Using & instead of && might introduce logic errors and bugs in your code that are difficult to find. Polyspace flags the use of bitwise operators in these kinds of expressions where relational operators are also used.

#### Correction — Use Logical Operators with Boolean-Like Operands

One possible correction is to use logical operators in expressions that contain relational operators and Boolean variables.

```
class User{
    //...
    int uid;
    int euid;
public:
    int getuid();
    int geteuid();
};
void Noncompliant ()
{
    User nU;
    if (nU.getuid () && nU.geteuid () == 0) {   //Compliant
        //...
    }else{
        //...
    }
}
```

## Check Information

**Category:** Behavioral Problems

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-480

# CWE Rule 481

Assigning instead of Comparing

## Description

### Rule Description

*The code uses an operator for assignment when the intention was to perform a comparison.*

### Polyspace Implementation

The rule checker checks for **Invalid use of = (assignment) operator**.

## Examples

### Invalid use of = (assignment) operator

#### Issue

This issue occurs when an assignment is made inside the predicate of a conditional, such as `if` or `while`.

In C and C++, a single equal sign is an assignment not a comparison. Using a single equal sign in a conditional statement can indicate a typo or a mistake.

#### Risk

- Conditional statement tests the wrong values— The single equal sign operation assigns the value of the right operand to the left operand. Then, because this assignment is inside the predicate of a conditional, the program checks whether the new value of the left operand is nonzero or not NULL.
- Maintenance and readability issues — Even if the assignment is intended, someone reading or updating the code can misinterpret the assignment as an equality comparison instead of an assignment.

#### Fix

- If the assignment is a bug, to check for equality, add a second equal sign (==).
- If the assignment inside the conditional statement was intentional, to improve readability, separate the assignment and the test. Move the assignment outside the control statement. In the control statement, simply test the result of the assignment.

  If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

  - “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
  - “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
  - “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example — Single Equal Sign Inside an if Condition

```
#include <stdio.h>

void bad_equals_ex(int alpha, int beta)
{
    if(alpha = beta)
    {
        printf("Equal\n");
    }
}
```

The equal sign is flagged as a defect because the assignment operator is used within the predicate of the if-statement. The predicate assigns the value `beta` to `alpha`, then implicitly tests whether `alpha` is true or false.

#### Correction — Change Expression to Comparison

One possible correction is adding an additional equal sign. This correction changes the assignment to a comparison. The if condition compares whether `alpha` and `beta` are equal.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha == beta)
    {
        printf("Equal\n");
    }
}
```

#### Correction — Assignment and Comparison Inside the `if` Condition

If an assignment must be made inside the predicate, a possible correction is adding an explicit comparison. This correction assigns the value of `beta` to `alpha`, then explicitly checks whether `alpha` is nonzero. The code is clearer.

```
#include <stdio.h>

int assignment_not_zero(int alpha, int beta)
{
    if((alpha = beta) != 0)
    {
        return alpha;
    }
    else
    {
        return 0;
    }
}
```

#### Correction — Move Assignment Outside the `if` Statement

If the assignment can be made outside the control statement, one possible correction is to separate the assignment and comparison. This correction assigns the value of `beta` to `alpha` before the if. Inside the if-condition, only `alpha` is given to test if `alpha` is nonzero or not NULL.

```
#include <stdio.h>
```

```
void assign_and_print(int alpha, int beta)
{
    alpha = beta;
    if(alpha)
    {
        printf("%d", alpha);
    }
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-481

# CWE Rule 482

Comparing instead of Assigning

## Description

### Rule Description

*The code uses an operator for comparison when the intention was to perform an assignment.*

### Polyspace Implementation

The rule checker checks for **Invalid use of == (equality) operator**.

## Examples

### Invalid use of == (equality) operator

#### Issue

This issue occurs when you use an equality operator instead of an assignment operator in a simple statement.

#### Risk

The use of == operator instead of an = operator can silently produce incorrect results. If you intended to assign a value to a variable, the assignment does not occur. The variable retains its previous value or if not initialized previously, stays uninitialized.

#### Fix

Use the = (assignment) operator instead of the == (equality) operator.

The check appears on chained assignment and equality operators such as:

```
compFlag = val1 == val2;
```

For better readability of your code, place the equality check in parenthesis.

```
compFlag = (val1 == val2);
```

If the use of == operator is intended, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Equality Evaluation in for-Loop

```
void populate_array(void)
{
    int i = 0;
    int j = 0;
    int array[4];

    for (j == 5; j < 9; j++) {
        array[i] = j;
        i++;
    }
}
```

Inside the `for`-loop, the statement `j == 5` tests whether `j` is equal to 5 instead of setting `j` to 5. The `for`-loop iterates from 0 to 8 because `j` starts with a value of 0, not 5. A by-product of the invalid equality operator is an out-of-bounds array access in the next line.

#### Correction — Change to Assignment Operator

One possible correction is to change the == operator to a single equal sign (=). Changing the == sign resolves both defects because the `for`-loop iterates the intended number of times.

```
void populate_array(void)
{
    int i = 0;
    int j = 0;
    int array[4];

    for (j = 5; j < 9; j++) {
        array[i] = j;
        i++;
    }
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-482

# CWE Rule 483

Incorrect Block Delimitation

## Description

### Rule Description

*The code does not explicitly delimit a block that is intended to contain 2 or more statements, creating a logic error.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Incorrectly indented statement**
- **Semicolon on the same line as an if, for or while statement**

## Examples

### Incorrectly indented statement

#### Issue

This issue occurs when the indentation of a statement makes it appear as part of an `if`, `else` or another block but the arrangement or lack of braces actually keeps the statement outside the block.

#### Risk

A developer or reviewer might incorrectly associate the statement with a block based on its indentation, leading to an incorrect assumption about the program logic.

For instance, in this example:

```
if(credentialsOK())
   login=1;
   setCookies();
```

the line `setCookies();` is not part of the `if` block, but the indentation suggests otherwise.

#### Fix

If you want a statement to be part of a block, make sure that the statement is within the braces associated with the block. To identify the extent of a block, on the **Source** pane, click the opening brace.

If an `if`, `else` or `while` statement has no braces following the condition, only the next line on an execution path up to a semicolon is considered part of the `if`, `else` or `while` block. If you want subsequent lines to be included in the block, wrap the lines in braces.

For instance, in the preceding example, to include both statements in the `if` block, use:

```
if(credentialsOK()) {
   login=1;
   setCookies();
}
```

**Example — else Statement Incorrectly Indented**

```
int switch1, switch2;

void doSomething(void);
void doSomethingElse(void);

void func() {
    if(switch1)
        if(switch2)
            doSomething();
    else
        doSomethingElse();
}
```

In this example, the `else` is indented as if it is associated with the first `if`. However, the `else` is actually associated with the second `if`. The indentation does not match the actual association and might lead to incorrect assumptions about the program logic.

**Correction – Use Braces Appropriately**

If you want the `else` to be associated with the first `if`, use braces to mark the boundaries of the first `if` block.

```
int switch1, switch2;

void doSomething(void);
void doSomethingElse(void);

void func() {
    if(switch1) {
        if(switch2)
            doSomething();
    }
    else
        doSomethingElse();
}
```

### Semicolon on the same line as an if, for or while statement

**Issue**

This issue occurs when a semicolon on the same line as the last token of an `if`, `for` or `while` statement results in an empty body.

The checker makes an exception for the case where the `if` statement is immediately followed by an `else` statement:

```
if(condition);
else {
  ...
}
```

#### Risk

The semicolon following the if, for or while statement often indicates a programming error. The spurious semicolon changes the execution flow and leads to unintended results.

#### Fix

If you want an empty body for the `if`, `for` or `while` statement, wrap the semicolon in a block and place the block on a new line to explicitly indicate your intent:

```
if(condition)
   {;}
```

Otherwise, remove the spurious semicolon.

#### Example — Spurious Semicolon

```
int credentialsOK(void);

void login () {
    int loggedIn = 0;
    if(credentialsOK());
      loggedIn = 1;
}
```

In this example, the spurious semicolon results in an empty `if` body. The assignment `loggedIn=1` is always performed. However, the assignment was probably to be performed only under a condition.

#### Correction – Remove Spurious Semicolon

If the semicolon was unintended, remove the semicolon.

```
int credentialsOK(void);

void login () {
    int loggedIn = 0;
    if(credentialsOK())
      loggedIn = 1;
}
```

## Check Information

**Category:** Behavioral Problems

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

CWE-483

# CWE Rule 484

Omitted Break Statement in Switch

## Description

### Rule Description

*The program omits a break statement within a switch or similar construct, causing code associated with multiple conditions to execute. This can cause problems when the programmer only intended to execute code associated with one condition.*

### Polyspace Implementation

The rule checker checks for **Missing break of switch case**.

## Examples

### Missing break of switch case

#### Issue

This issue occurs when a `case` block of `switch` statement does not end in a `break`, a `[[fallthrough]]`, or a code comment.

If the last entry in the case block is a code comment, for instance:

```
switch (wt)
    {
      case WE_W:
        do_something_for_WE_W();
        do_something_else_for_WE_W();
        /* fall through to WE_X*/
      case WE_X:
        ...
    }
```

Polyspace assumes that the missing break is intentional and does not raise a defect.

#### Risk

Switch cases without break statements fall through to the next switch case. If this fall-through is not intended, the switch case can unintentionally execute code and end the switch with unexpected results.

#### Fix

If you forgot the break statement, add it before the end of the switch case.

If you do not want a break for the highlighted switch case, add a comment to your code to document why this case falls through to the next case. This comment removes the defect from your results and makes your code more maintainable.

In some cases, such as in template functions, the defect checker might ignore the comment at the end of a case statement. A cleaner way to indicate a deliberate fall through to the next case is to use one of these `fallthrough` attributes if possible:

- `[[fallthrough]]`: Available since C++17.
- `[[gnu::fallthrough]]` and `[[clang::fallthrough]]`: Available with GCC and Clang compilers.
- `__attribute__((fallthrough))`: Available with GCC compilers.

**Example — Switch Without Break Statements**

```
enum WidgetEnum { WE_W, WE_X, WE_Y, WE_Z } widget_type;

extern void demo_do_something_for_WE_W(void);
extern void demo_do_something_for_WE_X(void);
extern void demo_report_error(void);

void bug_missingswitchbreak(enum WidgetEnum wt)
{
    /*
      In this non-compliant code example, the case where widget_type is WE_W lacks a
      break statement. Consequently, statements that should be executed only when
      widget_type is WE_X are executed even when widget_type is WE_W.
    */
    switch (wt)
    {
      case WE_W:
        demo_do_something_for_WE_W();
      case WE_X:
        demo_do_something_for_WE_X();
      default:
        /* Handle error condition */
        demo_report_error();
    }
}
```

In this example, there are two cases without `break` statements. When `wt` is `WE_W`, the statements for `WE_W`, `WE_X`, and the `default` case execute because the program falls through the two cases without a break. No defect is raised on the `default` case or last case because it does not need a break statement.

**Correction — Add a Comment or break**

To fix this example, either add a comment after the last statement in the case block and before the next case block to mark and document the acceptable fall-through or add a break statement to avoid fall-through. In this example, case `WE_W` is supposed to fall through, so a comment is added to explicitly state this action. For the second case, a break statement is added to avoid falling through to the `default` case.

```
enum WidgetEnum { WE_W, WE_X, WE_Y, WE_Z } widget_type;

extern void demo_do_something_for_WE_W(void);
extern void demo_do_something_for_WE_X(void);
extern void demo_report_error(void);

void corrected_missingswitchbreak(enum WidgetEnum wt)
{
    switch (wt)
    {
      case WE_W:
        demo_do_something_for_WE_W();
        /* fall through to WE_X*/
      case WE_X:
        demo_do_something_for_WE_X();
        break;
      default:
        /* Handle error condition */
        demo_report_error();
    }
}
```

## Check Information

**Category:** Behavioral Problems

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-484

# CWE Rule 489

Active Debug Code

## Description

### Rule Description

*The application is deployed to unauthorized actors with debugging code still enabled or active, which can create unintended entry points or expose sensitive information.*

### Polyspace Implementation

The rule checker checks for **Use of a forbidden function**.

## Examples

### Use of a forbidden function

#### Issue

This issue occurs when you use a function that appears in a blocklist of forbidden functions. To create the blocklist:

- List functions in an XML file in a specific syntax.

  Copy the template file `code-behavior-specifications-template.xml` from the folder *`polyspaceroot`*`\polyspace\verifier\cxx` to a writable location and modify the file. Enter each function in the file using the following syntax after existing similar entries:

  ```
  <function name="funcname">
     <behavior name="FORBIDDEN_FUNC"/>
  </function>
  ```

  where *`funcname`* is the name of the function you want to block.
- Specify this XML file as argument for the option `-code-behavior-specifications`. See “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers”.

#### Risk

A function might be blocked for one of these reasons:

- The function can lead to many situations where the behavior is undefined leading to security vulnerabilities, and a more secure function exists.

  You can forbid functions that are not explicitly checked by existing checkers such as `Use of dangerous standard function` or `Use of obsolete standard function`.
- The function is being deprecated as part of a migration, for instance, from C++98 to C++11.

  As part of a migration, you can make a list of functions that need to be replaced and use this checker to identify their use.

#### Fix

Replace the blocked function with an allowed function.

When rolling out this checker to a group, project or organization, create a list of blocked functions and their replacements so that results reviewers can consult the list and make appropriate replacements.

#### Extend Checker

This defect checker requires a blocklist of functions to be specified. Even if you specify the checker using the option `Find defects (-checkers)`, it is not enabled unless you also specify the blocklist. See “Flag Deprecated or Unsafe Functions, Keywords, or Macros Using Bug Finder Checkers”.

#### Example — Use of Forbidden Function std::signal

```
#include <csignal>
#include <iostream>

namespace
{
  volatile std::sig_atomic_t gSignalStatus;
}

void signal_handler(int signal)
{
  gSignalStatus = signal;
}

int main()
{
  // Install a signal handler
  std::signal(SIGINT, signal_handler);

  std::cout << "SignalValue: " << gSignalStatus << '\n';
  std::cout << "Sending signal " << SIGINT << '\n';
  std::raise(SIGINT);
  std::cout << "SignalValue: " << gSignalStatus << '\n';
}
```

Suppose you want to deprecate the `std::signal` function. Add the following to the template XML file after similar existing entries:

```
<specifications>
  <functions>
     <function name="std::signal">
        <behavior name="FORBIDDEN_FUNC"/>
     </function>
   </functions>
</specifications>
```

and specify the XML file with the option `-code-behavior-specifications`.

In the analysis results, all uses of the `std::signal` function are flagged by this checker.

**Example — Blocking C++ Overloaded Operators**

```
class orderedPair {
        int var1;
        int var2;
    public:
        orderedPair() {
            var1 = 0;
            var2 = 0;
        }
        orderedPair(int arg1, int arg2) {
            var1 = arg1;
            var2 = arg2;
        }
        orderedPair& operator=(const orderedPair& rhs) {
            var1 = rhs.var1;
            var2 = rhs.var2;
            return *this;
        }
        orderedPair& operator+(orderedPair& rhs) {
            var1 += rhs.var1;
            var2 += rhs.var2;
            return *this;
        }
};

void main() {
    int one=1, zero=0, sum;
    orderedPair firstOrderedPair(one, one);
    orderedPair secondOrderedPair(zero, one);
    orderedPair sumPair;

    sum = zero + one;
    sumPair = firstOrderedPair + secondOrderedPair;
}
```

Suppose you want to identify all the locations where operator overloads in the `orderedPair` class are used. Add the overloaded operators to the template XML file:

```
<specifications>
  <functions>
     <function name="orderedPair::operator=">
        <behavior name="FORBIDDEN_FUNC"/>
     </function>
     <function name="orderedPair::operator+">
        <behavior name="FORBIDDEN_FUNC"/>
     </function>
   </functions>
</specifications>
```

and specify the XML file with the option `-code-behavior-specifications`.

The analysis identifies all calls to the overloaded operators and flags their use. Using this method, you can distinguish specific overloads of an operator instead of searching for and browsing through all instances of the operator.

## Check Information

**Category:** Bad Coding Practices

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-489

# CWE Rule 495

Private Data Structure Returned From A Public Method

## Description

### Rule Description

*The product has a method that is declared public, but returns a reference to a private data structure, which could then be modified in unexpected ways.*

### Polyspace Implementation

The rule checker checks for **Return of non-const pointer or reference to private or protected data owned by the class**.

## Examples

### Return of non-const pointer or reference to private or protected data owned by the class

**Issue**

The issue occurs if a member function returns a non-`const` pointer or reference to a nonstatic data member. The rule does not apply to static data members.

**Risk**

Returning a nonconstant pointer or reference to private or protected class-owned data enables clients to externally access and modify the state of the object without an interface. Such access without an explicit interface might bypass the private/protected data access hierarchy of the class, which might result in unexpected behavior and lead to bugs.

This rule applies to data that is owned by the class. Nonconstant handles to objects that are shared between different classes might be returned. Classes that mimic smart pointers and containers do not violate this rule.

**Fix**

Do not return a non-`const` pointer or reference to class-owned data that is private or protected.

**Example — Avoid Returning Non-Constant Raw Pointers to Private Data**

```
#include <cstdint>
#include <memory>
#include <utility>

class A
{
  public:
    explicit A(std::int32_t number) : x(number) {}
    std::int32_t&
    GetX() noexcept // Noncompliant
    {
      return x;
    }

  private:
    std::int32_t x;
};

void Fn1() noexcept
{
  A a{10};
  std::int32_t& number = a.GetX();
  number = 15; // External modification of private class data
}
```

In this example, the `class A` member function `GetX()` returns a non-constant raw pointer to `x`, which is private data owned by `class A`. Polyspace flags this implementation as noncompliant. `Fn1()` demonstrates the issues of `a.GetX()` returning a non-constant raw pointer to private class data, which is then stored and modified by number. The class has no control over changes to its own private data member, which might lead to unexpected behavior.

**Example — Return Shared Smart Pointer Variables**

```
#include <cstdint>
#include <memory>
#include <utility>
class B
{
  public:
    explicit B(std::shared_ptr<std::int32_t> ptr) : sharedptr(std::move(ptr)) {}
    std::shared_ptr<std::int32_t> GetSharedPtr() const noexcept // Compliant
    {
      return sharedptr;
    }

  private:
    std::shared_ptr<std::int32_t> sharedptr;
};

void Fn2() noexcept
{
  std::shared_ptr<std::int32_t> ptr = std::make_shared<std::int32_t>(10);
  B b1{ptr};
  B b2{ptr};
  *ptr = 50; // External modification of ptr which shared between b1 and b2
  // instances
  auto shared = b1.GetSharedPtr();

  *shared = 100;   // External modification of ptr which shared between b1 and
  // b2 instances
}
```

In this example, the `class B` function `GetSharedPtr()` returns a smart pointer variable that is shared between the instances `b1` and `b2`. Polyspace does not flag this implementation as noncompliant.

**Example — Return Constant References**

```
#include <cstdint>
#include <memory>
#include <utility>
class C
{
  public:
    explicit C(std::int32_t number)
      : ownedptr{std::make_unique<std::int32_t>(number)}
    {
    }
    const std::int32_t& GetData() const noexcept // Compliant
    {
      return *ownedptr;
    }

  private:
    std::unique_ptr<std::int32_t> ownedptr;
};
void Fn3() noexcept
{
  C c{10};
  const std::int32_t& data = c.GetData();
  // data = 20; // Cannot modify data, it is a const reference
}
```

In this example, `GetData()` returns a constant reference. You cannot modify private class-data by using this member function. Polyspace does not flag this implementation as noncompliant.

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-495

# CWE Rule 500

Public Static Field Not Marked Final

## Description

### Rule Description

*An object contains a public static field that is not marked final, which might allow it to be modified in unexpected ways.*

### Polyspace Implementation

The rule checker checks for **Public static field not const**.

## Examples

### Public static field not const

#### Issue

This issue occurs when all of these conditions are true:

- A class contains a data member that is both `static` and `public`
- The `static` and `public` data member is not `const`

#### Risk

The `public static` data members of a class can be modified by any of class in the program. Unless `public static` data members are specified as `const`, they might be modified in unexpected ways and result in bugs that are difficult to diagnose.

#### Fix

To resolve this defect, specify the `public static` data members as `const` or `constexpr`.

#### Example — Avoid Nonconst public static Data Members

```
#include<string>
class SomeAppClass {

private:
    static std::string appPropertiesConfigFilep2;

protected:
    static std::string appPropertiesConfigFilepr2;

public:
    static std::string appPropertiesConfigFile2; //Defect

    static void foo() {}
    int bar() {
        static int v = 3;
        return v;
    }
};
```

In this example, the data member `appPropertiesConfigFile2` is `static` and `public`, but not `const`. Polyspace flags the member. The `static` data members that are `private` and `protected` are not flagged. This rule does not apply to `static` objects that are in the scope of `public` member functions of a class.

#### Correction — Declare `public static` Data Members as `const`

To resolve this defect, declare the `public static` data members of a class as `const`.

```
#include<string>
class SomeAppClass {

private:
    static std::string appPropertiesConfigFilep2;

protected:
    static std::string appPropertiesConfigFilepr2;

public:
    static const std::string appPropertiesConfigFile2; //No Defect

    static void foo() {}
    int bar() {
        static int v = 3;
        return v;
    }
};
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-500

# CWE Rule 522

Insufficiently Protected Credentials

## Description

### Rule Description

*The product transmits or stores authentication credentials, but it uses an insecure method that is susceptible to unauthorized interception and/or retrieval.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Constant cipher key**
- **Hard-coded sensitive data**
- **Nonsecure hash algorithm**
- **Nonsecure parameters for key generation**
- **Nonsecure RSA public exponent**
- **Nonsecure SSL/TLS protocol**
- **Unsafe standard encryption function**

## Examples

### Constant cipher key

#### Issue

This issue occurs when you use a constant for the encryption or decryption key.

#### Risk

If you use a constant for the encryption or decryption key, an attacker can retrieve your key easily.

You use a key to encrypt and later decrypt your data. If a key is easily retrieved, data encrypted using that key is not secure.

#### Fix

Produce a random key by using a strong random number generator.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

#### Example — Constants Used for Key

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16
```

```
int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16] = {'1', '2', '3', '4','5','6','b','8','9',
                                 '1','2','3','4','5','6','7'};
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the cipher key, `key`, has constants only. An attacker can easily retrieve a constant key.

#### Correction — Use Random Key

Use a strong random number generator to produce the cipher key. The corrected code here uses the function `RAND_bytes` declared in `openssl/rand.h`.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

### Hard-coded sensitive data

#### Issue

This issue occurs when data that is potentially sensitive is directly exposed in the code, for instance, as string literals. The checker identifies certain data as sensitive from their use in certain functions such as password encryption functions.

Following data can be potentially sensitive.

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Host name | • `sethostname`, `setdomainname`, `gethostbyname`, `gethostbyname2`, `getaddrinfo`, `gethostbyname_r`, `gethostbyname2_r` (string argument)<br>• `inet_aton`, `inet_pton`, `inet_net_pton`, `inet_addr`, `inet_network` (string argument)<br>• `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (2nd argument) |

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Password | • `CreateProcessWithLogonW`, `LogonUser` (1st argument)<br>• `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (3rd argument) |
| Database | • MySQL: `mysql_real_connect`, `mysql_real_connect_nonblocking`, `mysql_connect` (4th argument)<br>• SQLite: `sqlite3_open`, `sqlite3_open16`, `sqlite3_open_v2` (1st argument)<br>• PostgreSQL: `PQconnectdb`<br>• Microsoft SQL: `SQLDriverConnect` (3rd argument) |
| User name | • `getpw`, `getpwnam`, `getpwnam_r`, `getpwuid`, `getpwuid_r` |
| Salt | `crypt`, `crypt_r` (2nd argument) |
| Cryptography keys and initialization vectors | OpenSSL:<br>• `EVP_CipherInit`, `EVP_EncryptInit`, `EVP_DecryptInit` (3rd argument)<br>• `EVP_CipherInit_ex`, `EVP_EncryptInit_ex`, `EVP_DecryptInit_ex` (4th argument) |
| Seed | • `srand`, `srandom`, `initstate` (1st argument)<br>• OpenSSL: `RAND_seed`, `RAND_add` |

**Risk**

Information that is hardcoded can be queried from binaries generated from the code.

**Fix**

Avoid hard coding sensitive information.

**Example — Sensitive Data Exposed Through String Literals**

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;
```

```
// File that uses functions from "mysql.h"
const char *host = "localhost";
char *user = "guest";
char *passwd;

DbHandle connect_to_database_server(const char *db)
{
    passwd = (char*)"guest";
    return (DbHandle)
        mysql_real_connect (sql, host, user, passwd, db, 0, 0x0, 0);
}
```

In this example, the arguments `host` (host name), `user` (user name), and `passwd` (password) are string literals and directly exposed in the code.

Querying the generated binary for ASCII strings can reveal this information.

**Correction – Read Sensitive Data from Secured Configuration Files**

One possible correction is to read the data from a configuration file. In the following corrected example, the call to function `connect_to_database_server_init` presumably reads the host name, user name, and password into its arguments from a secured configuration file.

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"

int connect_to_database_server_init(const char **host,
                                    const char **user,
                                    const char **passwd,
                                    const char **db);

DbHandle connect_to_database_server(const char *db)
{
    const char *host_from_cfg;
    const char *user_from_cfg;
    const char *passwd_from_cfg;
    const char *db_from_cfg;
    if (connect_to_database_server_init(&host_from_cfg,
                                        &user_from_cfg,
                                        &passwd_from_cfg,
                                        &db_from_cfg))
    {
        return (DbHandle)
            mysql_real_connect (sql, host_from_cfg, user_from_cfg,
                        passwd_from_cfg, db_from_cfg,  0, 0x0, 0);
    }
    else
        return (DbHandle)0x0;
}
```

### Nonsecure hash algorithm

#### Issue

This issue occurs when you use a cryptographic hash function that is proven to be weak against certain forms of attack.

The hash functions flagged by this checker include SHA-0, SHA-1, MD4, MD5, and RIPEMD-160. The checker detects the use of these hash functions in:

- Functions from the EVP API such as `EVP_DigestUpdate` or `EVP_SignUpdate`.
- Functions from the low level API such as `SHA1_Update` or `MD5_Update`.

#### Risk

You use a hash function to create a message digest from input data and thereby ensure integrity of your data. The hash functions flagged by this checker use algorithms with known weaknesses that an attacker can exploit. The attacks can comprise the integrity of your data.

#### Fix

Use a more secure hash function. For instance, use the later SHA functions such as SHA-224, SHA-256, SHA-384, and SHA-512.

#### Example — Use of MD5 Algorithm

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
unsigned int out_len;

void func(unsigned char *src, size_t len, EVP_PKEY* pkey){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_md5(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_DigestUpdate(ctx,src,len);

  if (ret != 1) fatal_error();

  ret = EVP_SignFinal(ctx, out_buf, &out_len, pkey);
  if (ret != 1) fatal_error();
}
```

In this example, during initialization with `EVP_SignInit_ex`, the context object is associated with the weak hash function MD5. The checker flags the usage of this context in the update step with `EVP_DigestUpdate`.

#### Correction — Use SHA-2 Family Function

One possible correction is to use a hash function from the SHA-2 family, such as SHA-256.

```
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
unsigned char *out_buf;
unsigned int out_len;

void func(unsigned char *src, size_t len, EVP_PKEY* pkey){
  EVP_MD_CTX* ctx = EVP_MD_CTX_create();

  ret = EVP_SignInit_ex(ctx, EVP_sha256(), NULL);
  if (ret != 1) fatal_error();

  ret = EVP_SignUpdate(ctx, src, len);
  if (ret != 1) fatal_error();

  ret = EVP_SignFinal(ctx, out_buf, &out_len, pkey);
  if (ret != 1) fatal_error();
}
```

### Nonsecure parameters for key generation

#### Issue

This issue occurs when you attempt key generation by using an EVP_PKEY_CTX context object that is associated with weak parameters. What constitutes a weak parameter depends on the public key algorithm used. In the DSA algorithm, a weak parameter can be the result of setting an insufficient parameter length.

For instance, you set the number of bits used for DSA parameter generation to 512 bits, and then use the parameters for key generation:

```
EVP_PKEY_CTX *pctx,*kctx;
EVP_PKEY *params, *pkey;

/* Initializations for parameter generation */
pctx = EVP_PKEY_CTX_new_id(EVP_PKEY_DSA, NULL);
params = EVP_PKEY_new();

/* Parameter generation */
ret = EVP_PKEY_paramgen_init(pctx);
ret = EVP_PKEY_CTX_set_dsa_paramgen_bits(pctx, KEYLEN_512BITS);
ret = EVP_PKEY_paramgen(pctx, &params);

/* Initializations for key generation */
kctx = EVP_PKEY_CTX_new(params, NULL);
pkey = EVP_PKEY_new();

/* Key generation */
ret = EVP_PKEY_keygen_init(kctx);
ret = EVP_PKEY_keygen(kctx, &pkey);
```

#### Risk

Weak parameters lead to keys that are not sufficiently strong for encryption and expose sensitive information to known ways of attack.

#### Fix

Depending on the algorithm, use these parameters:

- Diffie-Hellman (DH): Set the length of the DH prime parameter to 2048 bits.

  ```
  ret = EVP_PKEY_CTX_set_dh_paramgen_prime_len(pctx, 2048);
  ```

  Set the DH generator to 2 or 5.

  ```
  ret = EVP_PKEY_CTX_set_dh_paramgen_generator(pctx, 2);
  ```
- Digital Signature Algorithm (DSA): Set the number of bits used for DSA parameter generation to 2048 bits.

  ```
  ret = EVP_PKEY_CTX_set_dsa_paramgen_bits(pctx, 2048);
  ```
- RSA: Set the RSA key length to 2048 bits.

  ```
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(kctx, 2048);
  ```
- Elliptic curve (EC): Avoid using curves that are known to be broken, for instance, `X9_62_prime256v1`. Use, for instance, `sect239k1`.

  ```
  ret = EVP_PKEY_CTX_set_ec_paramgen_curve_nid(pctx, NID_sect239k1);
  ```

#### Example — Insufficient Bits for RSA Key Generation

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  EVP_PKEY_CTX * ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 512);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

In this example, the RSA key generation uses 512 bits, which makes the generated key vulnerable to attacks.

#### Correction — Use 2048 bits

Use 2048 bits for RSA key generation.

```
#include <stddef.h>
```

```
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  EVP_PKEY_CTX * ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

### Nonsecure RSA public exponent

#### Issue

This issue occurs when you attempt RSA key generation by using a context object that is associated with a low public exponent.

For instance, you set a public exponent of 3 in the context object, and then use it for key generation.

```
/* Set public exponent */
ret = BN_dec2bn(&pubexp, "3");

/* Initialize context */
ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
pkey = EVP_PKEY_new();
ret = EVP_PKEY_keygen_init(kctx);

/* Set public exponent in context */
ret = EVP_PKEY_CTX_set_rsa_keygen_pubexp(ctx, pubexp);

/* Generate key */
ret = EVP_PKEY_keygen(kctx, &pkey);
```

#### Risk

A low RSA public exponent makes certain kinds of attacks more damaging, especially when a weak padding scheme is used or padding is not used at all.

#### Fix

It is recommended to use a public exponent of 65537. Using a higher public exponent can make the operations slower.

#### Example — Using RSA Public Exponent of 3

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)
```

```
int ret;
int func(EVP_PKEY *pkey){
  BIGNUM* pubexp;
  EVP_PKEY_CTX* ctx;

  pubexp = BN_new();
  if (pubexp == NULL) fatal_error();
  ret = BN_set_word(pubexp, 3);
  if (ret <= 0) fatal_error();

  ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_pubexp(ctx, pubexp);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

In this example, an RSA public exponent of 3 is associated with the context object `ctx`. The low exponent makes operations that use the generated key vulnerable to certain attacks.

**Correction — Use Public Exponent of 65537**

One possible correction is to use the recommended public exponent 65537.

```
#include <stddef.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#define fatal_error() exit(-1)

int ret;
int func(EVP_PKEY *pkey){
  BIGNUM* pubexp;
  EVP_PKEY_CTX* ctx;

  pubexp = BN_new();
  if (pubexp == NULL) fatal_error();
  ret = BN_set_word(pubexp, 65537);
  if (ret <= 0) fatal_error();

  ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
  if (ctx == NULL) fatal_error();

  ret = EVP_PKEY_keygen_init(ctx);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048);
  if (ret <= 0) fatal_error();
  ret = EVP_PKEY_CTX_set_rsa_keygen_pubexp(ctx, pubexp);
  if (ret <= 0) fatal_error();
  return EVP_PKEY_keygen(ctx, &pkey);
}
```

## Nonsecure SSL/TLS protocol

#### Issue

This issue occurs when you do not disable nonsecure protocols in an `SSL_CTX` or `SSL` context object before using the object for handling SSL/TLS connections.

For instance, you disable the protocols SSL2.0 and TLS1.0 but forget to disable the protocol SSL3.0, which is also considered weak.

```
/* Create and configure context */
ctx = SSL_CTX_new(SSLv23_method());
SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2|SSL_OP_NO_TLSv1);

/* Use context to handle connection */
ssl = SSL_new(ctx);
SSL_set_fd(ssl, NULL);
ret = SSL_connect(ssl);
```

#### Risk

The protocols SSL2.0, SSL3.0, and TLS1.0 are considered weak in the cryptographic community. Using one of these protocols can expose your connections to cross-protocol attacks. The attacker can decrypt an RSA ciphertext without knowing the RSA private key.

#### Fix

Disable the nonsecure protocols in the context object before using the object to handle connections.

```
/* Create and configure context */
ctx = SSL_CTX_new(SSLv23_method());
SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2|SSL_OP_NO_SSLv3|SSL_OP_NO_TLSv1);
```

#### Example — Nonsecure Protocols Not Disabled

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>


#define fatal_error() exit(-1)

int ret;
int func(){
  SSL_CTX *ctx;
  SSL *ssl;

  SSL_library_init();

  /* context configuration */
  ctx = SSL_CTX_new(SSLv23_client_method());
  if (ctx==NULL) fatal_error();

  ret = SSL_CTX_use_certificate_file(ctx, "cert.pem", SSL_FILETYPE_PEM);
  if (ret <= 0) fatal_error();

  ret = SSL_CTX_load_verify_locations(ctx, NULL, "ca/path");
  if (ret <= 0) fatal_error();

  /* Handle connection */
  ssl = SSL_new(ctx);
  if (ssl==NULL) fatal_error();
  SSL_set_fd(ssl, NULL);

  return SSL_connect(ssl);
}
```

In this example, the protocols SSL2.0, SSL3.0, and TLS1.0 are not disabled in the context object before the object is used for a new connection.

**Correction — Disable Nonsecure Protocols**

Disable nonsecure protocols before using the objects for a new connection. Use the function `SSL_CTX_set_options` to disable the protocols SSL2.0, SSL3.0, and TLS1.0.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>


#define fatal_error() exit(-1)

int ret;
int func(){
  SSL_CTX *ctx;
  SSL *ssl;

  SSL_library_init();

  /* context configuration */
  ctx = SSL_CTX_new(SSLv23_client_method());
  if (ctx==NULL) fatal_error();

  SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2|SSL_OP_NO_SSLv3|SSL_OP_NO_TLSv1);

  ret = SSL_CTX_use_certificate_file(ctx, "cert.pem", SSL_FILETYPE_PEM);
  if (ret <= 0) fatal_error();

  ret = SSL_CTX_load_verify_locations(ctx, NULL, "ca/path");
  if (ret <= 0) fatal_error();

  /* Handle connection */
  ssl = SSL_new(ctx);
  if (ssl==NULL) fatal_error();
  SSL_set_fd(ssl, NULL);
  return SSL_connect(ssl);
}
```

### Unsafe standard encryption function

#### Issue

This issue occurs when a standard encryption function uses a broken or weak cryptographic algorithm. For example, `crypt` is not reentrant and is based on the risky Data Encryption Standard (DES).

#### Risk

The use of a broken, weak, or nonstandard algorithm can expose sensitive information to an attacker. A determined hacker can access the protected data using various techniques.

If the weak function is nonreentrant, when you use the function in concurrent programs, there is an additional race condition risk.

#### Fix

Avoid functions that use these encryption algorithms. Instead, use a reentrant function that uses a stronger encryption algorithm.

**Note** Some implementations of `crypt` support additional, possibly more secure, encryption algorithms.

#### Example — Decrypting Password Using crypt

```
#define _GNU_SOURCE
#include <pwd.h>
#include <string.h>
#include <crypt.h>

volatile int rd = 1;

const char *salt = NULL;
struct crypt_data input, output;

int verif_pwd(const char *pwd, const char *cipher_pwd, int safe)
{
    int r = 0;
    char *decrypted_pwd = NULL;

    switch(safe)
    {
      case 1:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      case 2:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      default:
        decrypted_pwd = crypt(pwd, cipher_pwd);
```

```
            break;
    }

    r = (strcmp(cipher_pwd, decrypted_pwd) == 0);

    return r;
}
```

In this example, crypt_r and crypt decrypt a password. However, crypt is nonreentrant and uses the unsafe Data Encryption Standard algorithm.

**Correction — Use crypt_r**

One possible correction is to replace crypt with crypt_r.

```
#define _GNU_SOURCE
#include <pwd.h>
#include <string.h>
#include <crypt.h>

volatile int rd = 1;

const char *salt = NULL;
struct crypt_data input, output;

int verif_pwd(const char *pwd, const char *cipher_pwd, int safe)
{
    int r = 0;
    char *decrypted_pwd = NULL;

    switch(safe)
    {
      case 1:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      case 2:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;

      default:
        decrypted_pwd = crypt_r(pwd, cipher_pwd, &output);
        break;
    }

    r = (strcmp(cipher_pwd, decrypted_pwd) == 0);

    return r;
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

**Topics**

“Check for and Review Coding Standard Violations”

**External Websites**

CWE-522

# CWE Rule 547

Use of Hard-coded, Security-relevant Constants

## Description

### Rule Description

*The program uses hard-coded constants instead of symbolic names for security-critical values, which increases the likelihood of mistakes during code maintenance or security policy change.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Hard-coded buffer size**
- **Hard-coded loop boundary**
- **Hard-coded object size used to manipulate memory**
- **Hard-coded sensitive data**

## Examples

### Hard-coded buffer size

#### Issue

This issue occurs when you use a numerical value instead of a symbolic constant when declaring a memory buffer such as an array.

#### Risk

Hard-coded buffer size causes the following issues:

- Hard-coded buffer size increases the likelihood of mistakes and therefore maintenance costs. If a policy change requires developers to change the buffer size, they must change every occurrence of the buffer size in the code.
- Hard-constant constants can be exposed to attack if the code is disclosed.

#### Fix

Use a symbolic name instead of a hard-coded constant for buffer size. Symbolic names include `const`-qualified variables, `enum` constants, or macros.

`enum` constants are recommended.

- Macros are replaced by their constant values after preprocessing. Therefore, they can expose the loop boundary.
- `enum` constants are known at compilation time. Therefore, compilers can optimize the loops more efficiently.

  `const`-qualified variables are usually known at run time.

#### Example — Hard-Coded Buffer Size

```
int table[100];

void read(int);

void func(void) {
    for (int i=0; i<100; i++)
        read(table[i]);
}
```

In this example, the size of the array `table` is hard-coded.

#### Correction — Use Symbolic Name

One possible correction is to replace the hard-coded size with a symbolic name.

```
const int MAX_1 = 100;
#define MAX_2 100
enum { MAX_3 = 100 };

int table_1[MAX_1];
int table_2[MAX_2];
int table_3[MAX_3];

void read(int);

void func(void) {
    for (int i=0; i < MAX_1; i++)
        read(table_1[i]);
    for (int i=0; i < MAX_2; i++)
        read(table_2[i]);
    for (int i=0; i < MAX_3; i++)
        read(table_3[i]);
}
```

### Hard-coded loop boundary

#### Issue

This issue occurs when you use a numerical value instead of symbolic constant for the boundary of a `for`, `while` or `do-while` loop.

#### Risk

Hard-coded loop boundary causes the following issues:

- Hard-coded loop boundary makes the code vulnerable to denial of service attacks when the loop involves time-consuming computation or resource allocation.
- Hard-coded loop boundary increases the likelihood of mistakes and maintenance costs. If a policy change requires developers to change the loop boundary, they must change every occurrence of the boundary in the code.

  For instance, the loop boundary is 10000 and represents the maximum number of client connections supported in a network server application. If the server supports more clients, you must change all instances of the loop boundary in your code. Even if the loop boundary occurs once, you have to search for a numerical value of `10000` in your code. The numerical value can occur in places other than the loop boundary. You must browse through those places before you find the loop boundary.

#### Fix

Use a symbolic name instead of a hard-coded constant for loop boundary. Symbolic names include `const`-qualified variables, `enum` constants or macros.`enum` constants are recommended because:

- Macros are replaced by their constant values after preprocessing. Therefore, they can expose the buffer size.
- `enum` constants are known at compilation time. Therefore, compilers can allocate storage for them more efficiently.

  `const`-qualified variables are usually known at run time.

#### Example — Hard-Coded Loop Boundary

```
void performOperation(int);

void func(void) {
    for (int i=0; i<100; i++)
        performOperation(i);
}
```

In this example, the boundary of the `for` loop is hard-coded.

#### Correction — Use Symbolic Name

One possible correction is to replace the hard-coded loop boundary with a symbolic name.

```
const int MAX_1 = 100;
#define MAX_2 100
enum { MAX_3 = 100 };

void performOperation_1(int);
void performOperation_2(int);
void performOperation_3(int);

void func(void) {
    for (int i=0; i<MAX_1; i++)
        performOperation_1(i);
    for (int i=0; i<MAX_2; i++)
        performOperation_2(i);
    for (int i=0; i<MAX_3; i++)
        performOperation_3(i);
}
```

### Hard-coded object size used to manipulate memory

#### Issue

This issue occurs when you use hard-coded constants as memory size arguments for these memory functions:

- Dynamic memory allocation function such as `malloc` or `calloc`.
- Memory manipulation functions such as `memcpy`, `memmove`, `memcmp`, or `memset`.

When performing memory operations with a string literal, Polyspace does not report a defect if you hard code the memory size.

#### Risk

If you hard code object size, your code is not portable to architectures with different type sizes. If the constant value is not the same as the object size, the buffer might or might not overflow.

#### Fix

For the size argument of memory functions, use `sizeof(`*`object`*`)`.

#### Example — Assume 4-Byte Integer Pointers

```
#include <stddef.h>
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};
extern void fill_ints(int **matrix, size_t nb, size_t s);

void bug_hardcodedmemsize()
{
    size_t i, s;

    s = 4;
    int **matrix = (int **)calloc(SIZE20, s);
    if (matrix == NULL) {
        return; /* Indicate calloc() failure */
    }
    fill_ints(matrix, SIZE20, s);
    free(matrix);
}
```

In this example, the memory allocation function `calloc` is called with a memory size of 4. The memory is allocated for an integer pointer, which can be a more or less than 4 bytes depending on your target. If the integer pointer is not 4 bytes, your program can fail.

#### Correction — Use `sizeof(int *)`

When calling `calloc`, replace the hard-coded size with a call to `sizeof`. This change makes your code more portable.

```
#include <stddef.h>
#include <stdlib.h>
enum {
    SIZE3   = 3,
    SIZE20  = 20
};
extern void fill_ints(int **matrix, size_t nb, size_t s);

void corrected_hardcodedmemsize()
{
    size_t i, s;

    s = sizeof(int *);
    int **matrix = (int **)calloc(SIZE20, s);
    if (matrix == NULL) {
        return; /* Indicate calloc() failure */
    }
    fill_ints(matrix, SIZE20, s);
    free(matrix);
}
```

### Hard-coded sensitive data

#### Issue

This issue occurs when data that is potentially sensitive is directly exposed in the code, for instance, as string literals. The checker identifies certain data as sensitive from their use in certain functions such as password encryption functions.

Following data can be potentially sensitive.

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Host name | • sethostname, setdomainname, gethostbyname, gethostbyname2, getaddrinfo, gethostbyname_r, gethostbyname2_r (string argument)<br>• inet_aton, inet_pton, inet_net_pton, inet_addr, inet_network (string argument)<br>• mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (2nd argument) |
| Password | • CreateProcessWithLogonW, LogonUser (1st argument)<br>• mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (3rd argument) |
| Database | • MySQL: mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (4th argument)<br>• SQLite: sqlite3_open, sqlite3_open16, sqlite3_open_v2 (1st argument)<br>• PostgreSQL: PQconnectdb<br>• Microsoft SQL: SQLDriverConnect (3rd argument) |
| User name | • getpw, getpwnam, getpwnam_r, getpwuid, getpwuid_r |
| Salt | crypt, crypt_r (2nd argument) |

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Cryptography keys and initialization vectors | OpenSSL:<br>• EVP_CipherInit, EVP_EncryptInit, EVP_DecryptInit (3rd argument)<br>• EVP_CipherInit_ex, EVP_EncryptInit_ex, EVP_DecryptInit_ex (4th argument) |
| Seed | • srand, srandom, initstate (1st argument)<br>• OpenSSL: RAND_seed, RAND_add |

**Risk**

Information that is hardcoded can be queried from binaries generated from the code.

**Fix**

Avoid hard coding sensitive information.

**Example — Sensitive Data Exposed Through String Literals**

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"
const char *host = "localhost";
char *user = "guest";
char *passwd;

DbHandle connect_to_database_server(const char *db)
{
    passwd = (char*)"guest";
    return (DbHandle)
        mysql_real_connect (sql, host, user, passwd, db, 0, 0x0, 0);
}
```

In this example, the arguments host (host name), user (user name), and passwd (password) are string literals and directly exposed in the code.

Querying the generated binary for ASCII strings can reveal this information.

### Correction – Read Sensitive Data from Secured Configuration Files

One possible correction is to read the data from a configuration file. In the following corrected example, the call to function `connect_to_database_server_init` presumably reads the host name, user name, and password into its arguments from a secured configuration file.

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"

int connect_to_database_server_init(const char **host,
                                    const char **user,
                                    const char **passwd,
                                    const char **db);

DbHandle connect_to_database_server(const char *db)
{
    const char *host_from_cfg;
    const char *user_from_cfg;
    const char *passwd_from_cfg;
    const char *db_from_cfg;
    if (connect_to_database_server_init(&host_from_cfg,
                                        &user_from_cfg,
                                        &passwd_from_cfg,
                                        &db_from_cfg))
    {
        return (DbHandle)
            mysql_real_connect (sql, host_from_cfg, user_from_cfg,
                        passwd_from_cfg, db_from_cfg,  0, 0x0, 0);
    }
    else
        return (DbHandle)0x0;
}
```

## Check Information

**Category:** Bad Coding Practices

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-547

# CWE Rule 558

Use of getlogin() in Multithreaded Application

## Description

### Rule Description

*The application uses the getlogin() function in a multithreaded context, potentially causing it to return incorrect values.*

### Polyspace Implementation

The rule checker checks for **Unsafe standard function**.

## Examples

### Unsafe standard function

**Issue**

This issue occurs when you use standard functions that are unsafe and must not be used for security-related programming. Functions can be unsafe for many reasons. Some functions are unsafe because they are nonreentrant. Other functions change behavior depending on the target or platform, making some implementations unsafe.

**Risk**

Some unsafe functions are not reentrant, meaning that the contents of the function are not locked during a call. So, an attacker can change the values midstream.

`getlogin` specifically can be unsafe depending on the implementation. Some implementations of `getlogin` return only the first eight characters of a log-in name. An attacker can use a different login with the same first eight characters to gain entry and manipulate the program.

**Fix**

Avoid unsafe functions for security-related purposes. If you cannot avoid unsafe functions, use a safer version of the function instead. For `getlogin`, use `getlogin_r`.

**Example — Using getlogin**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <string.h>
#include <stdlib.h>


volatile int rd = 1;

int login_name_check(char *user)
{
    int r = -2;
    char *name = getlogin();
    if (name != NULL)
    {
        if (strcmp(name, user) == 0)
        {
            r = 0;
        }
        else
            r = -1;
    }

    return r;
}
```

This example uses `getlogin` to compare the user name of the current user to the given user name . However, `getlogin` can return something other than the current user name because a parallel process can change the string.

**Correction — Use `getlogin_r`**

One possible correction is to use `getlogin_r` instead of `getlogin`. `getlogin_r` is reentrant, so you can trust the result.

```
#define _POSIX_C_SOURCE 199506L // use of getlogin_r
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <string.h>
#include <stdlib.h>


volatile int rd = 1;

enum {  NAME_MAX_SIZE=64  };

int login_name_check(char *user)
{
    int r;
    char name[NAME_MAX_SIZE];

    if (getlogin_r(name, sizeof(name)) == 0)
    {
        if ((strlen(user) < sizeof(name)) &&
                     (strncmp(name, user, strlen(user)) == 0))
        {
            r = 0;
        }
        else
            r = -1;
    }
    else
        r = -2;
    return r;
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-558

# CWE Rule 560

Use of umask() with chmod-style Argument

## Description

### Rule Description

*The product calls umask() with an incorrect argument that is specified as if it is an argument to chmod().*

### Polyspace Implementation

The rule checker checks for **Umask used with chmod-style arguments**.

## Examples

### Umask used with chmod-style arguments

#### Issue

This issue occurs when `umask` commands have arguments specified in the style of arguments to `chmod` and provide possibly unintended permissions. For instance:

- The `umask` command provides more permissions to the group than the current user.
- The `umask` command provides more permissions to other users than the group.

For new files, the `umask` argument or the mask value specifies which permissions *not* to set, in other words, which permissions to remove. The mask is bitwise-negated and then applied to new file permissions. In contrast, `chmod` sets the permissions as you specify them.

#### Risk

If you use `chmod`-style arguments, you specify opposite permissions of what you want. This mistake can give external users unintended read/write access to new files and folders.

#### Fix

To fix this defect, perform both of these tasks:

- Enable read permission for the user.
- Set the mask value so that the user (`u`) has equal or fewer permissions turned off than the group (`g`) and the group has equal or fewer permissions turned off than other users (`o`), or `u <= g <= o`.

You can see the umask value by calling,

```
umask
```

or the symbolic value by calling,

```
umask -S
```

**Example — Setting the Default Mask**

```
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>

typedef mode_t (*umask_func)(mode_t);

const mode_t default_mode = (
    S_IRUSR    /* 00400 */
    | S_IWUSR  /* 00200 */
    | S_IRGRP  /* 00040 */
    | S_IWGRP  /* 00020 */
    | S_IROTH  /* 00004 */
    | S_IWOTH  /* 00002 */
    );         /* 00666 (i.e. -rw-rw-rw-) */

static void my_umask(mode_t mode)
{
    umask(mode);
}

int umask_use(mode_t m)
{
    my_umask(default_mode);
    return 0;
}
```

This example uses a function called `my_umask` to set the default mask mode. However, the `default_mode` variable gives the permissions 666 or `-rw-rw-rw`. `umask` negates this value. However, this negation means the default mask mode turns off read/write permissions for the user, group users, and other outside users.

**Correction — Negate Preferred Permissions**

One possible correction is to negate the `default_mode` argument to `my_umask`. This correction nullifies the negation `umask` for new files.

```
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>

typedef mode_t (*umask_func)(mode_t);

const mode_t default_mode = (
    S_IRUSR    /* 00400 */
    | S_IWUSR  /* 00200 */
    | S_IRGRP  /* 00040 */
    | S_IWGRP  /* 00020 */
    | S_IROTH  /* 00004 */
    | S_IWOTH  /* 00002 */
    );         /* 00666 (i.e. -rw-rw-rw-) */

static void my_umask(mode_t mode)
{
    umask(mode);
}

int umask_use(mode_t m)
{
    my_umask(~default_mode);
    return 0;
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-560

# CWE Rule 561

Dead Code

## Description

### Rule Description

*The software contains dead code, which can never be executed.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Dead code**
- **Static uncalled function**
- **Unreachable code**

## Examples

### Dead code

#### Issue

This issue occurs when a block of code cannot be reached because of a condition that is always true or false. This defect excludes:

- `Code deactivated by constant false condition`, which checks for directives with compile-time constants such as `#if 0` or `if(0)`.
- `Unreachable code`, which checks for code after a control escape such as `goto`, `break`, or `return`.
- `Useless if`, which checks for if statements that are always true.

#### Risk

Dead code wastes development time, memory and execution cycles. Developers have to maintain code that is not being executed. Instructions that are not executed still have to be stored and cached.

Dead code often represents legacy code that is no longer used. Cleaning up dead code periodically reduces future maintenance.

#### Fix

The fix depends on the root cause of the defect. For instance, the root cause can be an error condition that is checked twice on the same execution path, making the second check redundant and the corresponding block dead code.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder

Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you see dead code from use of functions such as `isinf` and `isnan`, enable an analysis mode that takes into account non-finite values. See `Consider non finite floats (-allow-non-finite-floats)`.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example — Dead Code from if-Statement**

```
#include <stdio.h>

int Return_From_Table(int ch){

    int table[5];

    /* Create a table */
    for(int i=0;i<=4;i++){
        table[i]=i^2+i+1;
    }

    if(table[ch]>100){ /* Defect: Condition always false */
         return 0;
    }
    return table[ch];
}
```

The maximum value in the array `table` is `4^2+4+1=21`, so the test expression `table[ch]>100` always evaluates to false. The `return 0` in the `if` statement is not executed.

**Correction — Remove Dead Code**

One possible correction is to remove the `if` condition from the code.

```
#include <stdio.h>

int Return_From_Table(int ch){

    int table[5];

    /* Create a table */
    for(int i=0;i<=4;i++){
        table[i]=i^2+i+1;
    }

    return table[ch];
}
```

#### Example — Dead Code for if with Enumerated Type

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;

    if (card > 7) {
        do_something(card);
    }
}
```

The type `suit` is enumerated with five options. However, the conditional expression `card > 7` always evaluates to false because `card` can be at most 5. The content in the `if` statement is not executed.

#### Correction — Change Condition

One possible correction is to change the if-condition in the code. In this correction, the 7 is changed to `HEART` to relate directly to the type of `card`.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;

    if (card > HEARTS) {
        do_something(card);
    }
}
```

### Static uncalled function

#### Issue

This issue occurs when a `static` function is not called in the same file where it is defined.

#### Risk

Uncalled functions often result from legacy code and cause unnecessary maintenance.

#### Fix

If the function is not meant to be called, remove the function. If the function is meant for debugging purposes only, wrap the function definition in a debug macro.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example — Uncalled function error**

Save the following code in the file `Initialize_Value.c`

```
#include <stdlib.h>
#include <stdio.h>

static int Initialize(void)
/* Defect: Function not called */
  {
   int input;
   printf("Enter an integer:");
   scanf("%d",&input);
   return(input);
  }

 void main()
  {
   int num;

   num=0;

   printf("The value of num is %d",num);
  }
```

The `static` function `Initialize` is not called in the file `Initialize_Value.c`.

**Correction — Call Function at Least Once**

One possible correction is to call `Initialize` at least once in the file `Initialize_Value.c`.

```
#include <stdlib.h>
#include <stdio.h>

static int Initialize(void)
  {
   int input;
   printf("Enter an integer:");
   scanf("%d",&input);
   return(input);
  }

 void main()
  {
   int num;

   /* Fix: Call static function Initialize */
   num=Initialize();

   printf("The value of num is %d",num);
  }
```

## Unreachable code

### Issue

This issue occurs when a section of code cannot be reached because of a previous break in control flow.

Statements such as `break`, `goto`, and `return`, move the flow of the program to another section or function. Because of this flow escape, the statements following the control-flow code, statistically, do not execute, and therefore the statements are unreachable.

This check also finds code following trivial infinite loops, such as `while(1)`. These types of loops only release the flow of the program by exiting the program. This type of exit causes code after the infinite loop to be unreachable.

### Risk

Unreachable code wastes development time, memory and execution cycles. Developers have to maintain code that is not being executed. Instructions that are not executed still have to be stored and cached.

### Fix

The fix depends on the intended functionality of the unreachable code. If you want the code to be executed, check the placement of the code or the prior statement that diverts the control flow. For instance, if the unreachable code follows a `return` statement, you might have to switch their order or remove the `return` statement altogether.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

### Example — Unreachable Code After Return

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if( (card < SPADES) || (card > CLUBS) )
        card = UNKNOWN_SUIT;
        return card;

    if (card < HEARTS) {
        guess(card);
    }
    return card;
}
```

In this example, there are missing braces and misleading indentation. The first `return` statement changes the flow of code back to where the function was called. Because of this return statement, the `if`-block and second `return` statement do not execute.

If you correct the indentation and the braces, the error becomes clearer.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if( (card < SPADES) || (card > CLUBS) ){
        card = UNKNOWN_SUIT;
    }
    return card;

    if (card < HEARTS) {
        guess(card);
    }
    return card;
}
```

#### Correction — Remove Return

One possible correction is to remove the escape statement. In this example, remove the first `return` statement to reach the final `if` statement.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if( (card < SPADES) || (card > CLUBS) )
    {
        card = UNKNOWN_SUIT;
    }

    if(card < HEARTS)
    {
        guess(card);
    }
    return card;
}
```

#### Correction — Remove Unreachable Code

Another possible correction is to remove the unreachable code if you do not need it. Because the function does not reach the second `if`-statement, removing it simplifies the code and does not change the program behavior.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if( (card < SPADES) || (card > CLUBS) )
    {
        card = UNKNOWN_SUIT;
    }
    return card;
}
```

**Example — Infinite Loop Causing Unreachable Code**

```
int add_apples(int apple) {
    int count = 1;
    while(1) {
        if(apple < 99){
            apple++;
            count++;
        }else{
            count--;
        }
    }
    return count;
}
```

In this example, the `while(1)` statement creates an infinite loop. The `return count` statement following this infinite loop is unreachable because the only way to exit this infinite loop is to exit the program.

**Correction — Rewrite Loop Condition**

One possible correction is to change the loop condition to make the `while` loop finite. In the example correction here, the loop uses the statement from the `if` condition: `apple < 99`.

```
int add_apples1(int apple) {
    int count = 0;
    while(apple < 99) {
        apple++;
        count++;
    }
    if(count == 0)
        count = -1;
    return count;
}
```

**Correction — Add a Break Statement**

Another possible correction is to add a break from the infinite loop, so there is a possibility of reaching code after the infinite loop. In this example, a `break` is added to the `else` block making the `return count` statement reachable.

```
int add_apples(int apple) {
    int count = 1;
    while(1) {
        if(apple < 99)
        {
            apple++;
            count++;
        }else{
            count--;
            break;
        }
    }
    return count;
}
```

#### Correction — Remove Unreachable Code

Another possible correction is to remove the unreachable code. This correction cleans up the code and makes it easier to review and maintain. In this example, remove the return statement and change the function return type to `void`.

```
void add_apples(int apple) {
    int count = 1;
    while(1) {
        if(apple < 99)
        {
            apple++;
            count++;
        }else{
            count--;
        }
    }
}
```

## Check Information

**Category:** Bad Coding Practices

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-561

# CWE Rule 562

Return of Stack Variable Address

## Description

### Rule Description

*A function returns the address of a stack variable, which will cause unintended program behavior, typically in the form of a crash.*

### Polyspace Implementation

The rule checker checks for **Pointer or reference to stack variable leaving scope**.

## Examples

### Pointer or reference to stack variable leaving scope

#### Issue

This issue occurs when a pointer or reference to a local variable leaves the scope of the variable. For instance:

- A function returns a pointer to a local variable.
- A function performs the assignment `globPtr = &locVar`. `globPtr` is a global pointer variable and `locVar` is a local variable.
- A function performs the assignment `*paramPtr = &locVar`. `paramPtr` is a function parameter that is, for instance, an `int**` pointer and `locVar` is a local `int` variable.
- A C++ method performs the assignment `memPtr = &locVar`. `memPtr` is a pointer data member of the class the method belongs to. `locVar` is a variable local to the method.
- (C++11 and later) A function returns a lambda expression object that captures local variables of the function by reference.

The defect also applies to memory allocated using the `alloca` function. The defect does not apply to static, local variables. Polyspace assumes that the local objects within a function definition are in the same scope.

#### Risk

Local variables are allocated an address on the stack. Once the scope of a local variable ends, this address is available for reuse. Using this address to access the local variable value outside the variable scope can cause unexpected behavior.

If a pointer to a local variable leaves the scope of the variable, Polyspace Bug Finder highlights the defect. The defect appears even if you do not use the address stored in the pointer. For maintainable code, it is a good practice to not allow the pointer to leave the variable scope. Even if you do not use the address in the pointer now, someone else using your function can use the address, causing undefined behavior.

### Fix

Do not allow a pointer or reference to a local variable to leave the variable scope.

### Example — Pointer to Local Variable Returned from Function

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In this example, `func1` returns a pointer to local variable `ret`.

In `main`, `ptr` points to the address of the local variable. When `ptr` is accessed in `func2`, the access is illegal because the scope of `ret` is limited to `func1`,

### Example — Pointer to Local Variable Escapes Through Lambda Expression

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [&] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

In this example, the `createAdder` function defines a lambda expression `adder` that captures the local variable `addThis` by reference. The scope of `addThis` is limited to the `createAdder` function. When the object returned by `createAdder` is called, a reference to the variable `addThis` is accessed outside its scope. When accessed in this way, the value of `addThis` is undefined.

### Correction – Capture Local Variables by Copy in Lambda Expression Instead of Reference

If a function returns a lambda expression object, avoid capturing local variables by reference in the lambda object. Capture the variables by copy instead.

Variables captured by copy have the same lifetime as the lambda object, but variables captured by reference often have a smaller lifetime than the lambda object itself. When the lambda object is used, these variables accessed outside scope have undefined values.

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [=] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

### Check Information

**Category:** Bad Coding Practices

## Version History

**Introduced in R2023a**

### See Also

`Check CWE (-cwe)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

CWE-562

# CWE Rule 563

Assignment to Variable without Use

## Description

### Rule Description

*The variable's value is assigned but never used, making it a dead store.*

### Polyspace Implementation

The rule checker checks for **Write without a further read**.

## Examples

### Write without a further read

#### Issue

This issue occurs when a value assigned to a variable is never read.

For instance, you write a value to a variable and then write a second value before reading the previous value. The first write operation is redundant.

#### Risk

Redundant write operations often indicate programming errors. For instance, you forgot to read the variable between two successive write operations or unintentionally read a different variable.

#### Fix

Identify the reason why you write to the variable but do not read it later. Look for common programming errors such as accidentally reading a different variable with a similar name.

If you determine that the write operation is redundant, remove the operation.

#### Example — Write Without Further Read Error

```
void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();
    /* Defect: Useless write */
}
```

After the variable `level` gets assigned the value `4 * getsensor()`, it is not read.

#### Correction — Use Value After Assignment

One possible correction is to use the variable `level` after the assignment.

```
#include <stdio.h>

void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();

    /* Fix: Use level after assignment */
    printf("The value is %d", level);

}
```

The variable `level` is printed, reading the new value.

## Check Information

**Category:** Bad Coding Practices

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-563

# CWE Rule 570

Expression is Always False

## Description

### Rule Description

*The software contains an expression that will always evaluate to false.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Code deactivated by constant false condition**
- **Dead code**

## Examples

### Code deactivated by constant false condition

#### Issue

This issue occurs when a block of code is deactivated using a `#if 0` directive or `if(0)` condition.

#### Risk

A `#if 0` directive or `if(0)` condition is used to temporarily deactivate segments of code. If your production code contains these directives, it means that the deactivation has not been lifted before shipping the code.

#### Fix

If the segment of code is present for debugging purposes only, remove the segment from production code. If the deactivation occurred by accident, remove the `#if 0` and `#endif` statements.

Often, a segment of code is deactivated for specific conditions, for instance, a specific operating system. Use macros with the `#if` directive to indicate these conditions instead of deactivating the code completely with a `#if 0` directive. For instance, GCC provides macros to detect the Windows operating system:

```
#ifdef _WIN32
   //Code deactivated for all operating systems
   //Other than 32-bit Windows
#endif
```

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.

- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example — Code Deactivated by Constant False Condition Error**

```
#include<stdio.h>
int Trim_Value(int* Arr,int Size,int Cutoff)
{
    int Count=0;

    for(int i=0;i < Size;i++){
        if(Arr[i]>Cutoff){
            Arr[i]=Cutoff;
            Count++;
        }
    }

    #if 0
    /* Defect: Code Segment Deactivated */

    if(Count==0){
        printf("Values less than cutoff.");
    }
     #endif

    return Count;
}
```

In the preceding code, the `printf` statement is placed within a `#if #endif` directive. The software treats the portion within the directive as code comments and not compiled.

**Correction — Change `#if 0` to `#if 1`**

Unless you intended to deactivate the `printf` statement, one possible correction is to reactivate the block of code in the `#if #endif` directive. To reactivate the block, change `#if 0` to `#if 1`.

```
#include<stdio.h>
int Trim_Value(int* Arr,int Size,int Cutoff)
{
 int Count=0;

 for(int i=0;i < Size;i++)
     {
      if(Arr[i]>Cutoff)
           {
            Arr[i]=Cutoff;
            Count++;
           }
     }


 /* Fix: Replace #if 0 by #if 1 */
 #if 1
      if(Count==0)
           {
            printf("Values less than cutoff.");
           }
 #endif
 return Count;
}
```

### Dead code

#### Issue

This issue occurs when a block of code cannot be reached because of a condition that is always true or false. This defect excludes:

- `Code deactivated by constant false condition`, which checks for directives with compile-time constants such as `#if 0` or `if(0)`.
- `Unreachable code`, which checks for code after a control escape such as `goto`, `break`, or `return`.
- `Useless if`, which checks for if statements that are always true.

#### Risk

Dead code wastes development time, memory and execution cycles. Developers have to maintain code that is not being executed. Instructions that are not executed still have to be stored and cached.

Dead code often represents legacy code that is no longer used. Cleaning up dead code periodically reduces future maintenance.

#### Fix

The fix depends on the root cause of the defect. For instance, the root cause can be an error condition that is checked twice on the same execution path, making the second check redundant and the corresponding block dead code.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you see dead code from use of functions such as `isinf` and `isnan`, enable an analysis mode that takes into account non-finite values. See `Consider non finite floats (-allow-non-finite-floats)`.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example — Dead Code from if-Statement

```
#include <stdio.h>
```

```
int Return_From_Table(int ch){

    int table[5];

    /* Create a table */
    for(int i=0;i<=4;i++){
        table[i]=i^2+i+1;
    }

    if(table[ch]>100){ /* Defect: Condition always false */
         return 0;
    }
    return table[ch];
}
```

The maximum value in the array `table` is `4^2+4+1=21`, so the test expression `table[ch]>100` always evaluates to false. The `return 0` in the `if` statement is not executed.

**Correction — Remove Dead Code**

One possible correction is to remove the `if` condition from the code.

```
#include <stdio.h>

int Return_From_Table(int ch){

    int table[5];

    /* Create a table */
    for(int i=0;i<=4;i++){
        table[i]=i^2+i+1;
    }

    return table[ch];
}
```

**Example — Dead Code for if with Enumerated Type**

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;

    if (card > 7) {
        do_something(card);
    }
}
```

The type `suit` is enumerated with five options. However, the conditional expression `card > 7` always evaluates to false because `card` can be at most 5. The content in the `if` statement is not executed.

#### Correction — Change Condition

One possible correction is to change the if-condition in the code. In this correction, the 7 is changed to HEART to relate directly to the type of `card`.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;

    if (card > HEARTS) {
        do_something(card);
    }
}
```

## Check Information

**Category:** Expression Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-570

# CWE Rule 571

Expression is Always True

## Description

### Rule Description

*The software contains an expression that will always evaluate to true.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Dead code**
- **Useless if**

## Examples

### Dead code

#### Issue

This issue occurs when a block of code cannot be reached because of a condition that is always true or false. This defect excludes:

- `Code deactivated by constant false condition`, which checks for directives with compile-time constants such as `#if 0` or `if(0)`.
- `Unreachable code`, which checks for code after a control escape such as `goto`, `break`, or `return`.
- `Useless if`, which checks for if statements that are always true.

#### Risk

Dead code wastes development time, memory and execution cycles. Developers have to maintain code that is not being executed. Instructions that are not executed still have to be stored and cached.

Dead code often represents legacy code that is no longer used. Cleaning up dead code periodically reduces future maintenance.

#### Fix

The fix depends on the root cause of the defect. For instance, the root cause can be an error condition that is checked twice on the same execution path, making the second check redundant and the corresponding block dead code.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you see dead code from use of functions such as `isinf` and `isnan`, enable an analysis mode that takes into account non-finite values. See `Consider non finite floats (-allow-non-finite-floats)`.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

**Example — Dead Code from if-Statement**

```
#include <stdio.h>

int Return_From_Table(int ch){

    int table[5];

    /* Create a table */
    for(int i=0;i<=4;i++){
        table[i]=i^2+i+1;
    }

    if(table[ch]>100){ /* Defect: Condition always false */
         return 0;
    }
    return table[ch];
}
```

The maximum value in the array `table` is `4^2+4+1=21`, so the test expression `table[ch]>100` always evaluates to false. The `return 0` in the `if` statement is not executed.

**Correction — Remove Dead Code**

One possible correction is to remove the `if` condition from the code.

```
#include <stdio.h>

int Return_From_Table(int ch){

    int table[5];

    /* Create a table */
    for(int i=0;i<=4;i++){
        table[i]=i^2+i+1;
    }

    return table[ch];
}
```

**Example — Dead Code for if with Enumerated Type**

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
```

```
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;

    if (card > 7) {
        do_something(card);
    }
}
```

The type `suit` is enumerated with five options. However, the conditional expression `card > 7` always evaluates to false because `card` can be at most 5. The content in the `if` statement is not executed.

**Correction — Change Condition**

One possible correction is to change the if-condition in the code. In this correction, the 7 is changed to `HEART` to relate directly to the type of `card`.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;

    if (card > HEARTS) {
        do_something(card);
    }
}
```

## Useless if

**Issue**

This issue occurs on `if`-statements where the condition is always true. This issue occurs only on if-statements that do not have an else-statement.

This defect shows unnecessary `if`-statements when there is no difference in code execution if the `if`-statement is removed.

**Risk**

Unnecessary `if` statements often indicate a coding error. Perhaps the `if` condition is coded incorrectly or the `if` statement is not required at all.

**Fix**

The fix depends on the root cause of the defect. For instance, the root cause can be an error condition that is checked twice on the same execution path, making the second check redundant.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If the redundant condition represents defensive coding practices and you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example — if with Enumerated Type

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card < 7) {
        do_something(card);
    }
}
```

The type `suit` is enumerated with five options. However, the conditional expression `card < 7` always evaluates to true because `card` can be at most 5. The `if` statement is unnecessary.

#### Correction — Change Condition

One possible correction is to change the if-condition in the code. In this correction, the 7 is changed to `UNKNOWN_SUIT` to relate directly to the type of `card`.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card > UNKNOWN_SUIT) {
        do_something(card);
    }
}
```

#### Correction — Remove If

Another possible correction is to remove the if-condition in the code. Because the condition is always true, you can remove the condition to simplify your code.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    do_something(card);
}
```

## Check Information

**Category:** Expression Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-571

# CWE Rule 587

Assignment of a Fixed Address to a Pointer

## Description

### Rule Description

*The software sets a pointer to a specific address other than NULL or 0.*

### Polyspace Implementation

The rule checker checks for **Function pointer assigned with absolute address**.

## Examples

### Function pointer assigned with absolute address

**Issue**

This issue occurs when a function pointer is assigned an absolute address.

Bug Finder considers expressions with any combination of literal constants as an absolute address. The one exception is when the value of the expression is zero.

**Risk**

Using a fixed address is not portable because it is possible that the address is invalid on other platforms.

An attacker can inject code at the absolute address, causing your program to execute arbitrary, possibly malicious, code.

**Fix**

Do not use an absolute address with function pointers.

**Example — Function Pointer Address Assignment**

```
extern int func0(int i, char c);
typedef int (*FuncPtr) (int, char);

FuncPtr funcptrabsoluteaddr() {
    return (FuncPtr)0x08040000;
}
```

In this example, the function returns a function pointer to the address `0x08040000`. If an attacker knows this absolute address, an attacker can compromise your program.

**Correction — Function Address**

One possible correction is to use the address of an existing function instead.

```
extern int func0(int i, char c);
typedef int (*FuncPtr) (int, char);
```

```
FuncPtr funcptrabsoluteaddr() {
    return &func0;
}
```

## Check Information

**Category:** Pointer Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-587

# CWE Rule 617

Reachable Assertion

## Description

### Rule Description

*The product contains an assert() or similar statement that can be triggered by an attacker, which leads to an application exit or other behavior that is more severe than necessary.*

### Polyspace Implementation

The rule checker checks for **Assertion**.

## Examples

### Assertion

**Issue**

This issue occurs when you use an `assert`, and the asserted expression is or could be false.

**Note** Polyspace does not flag `assert(0)` as an assertion defect because these statements are commonly used to disable certain sections of code.

**Risk**

Typically you use `assert` statements for functional testing in debug mode. An assertion failure found using static analysis indicates that the corresponding functional test would fail at run time.

**Fix**

The fix depends on the root cause of the defect. For instance, the root cause can be unconstrained input from an external source that eventually led to the assertion failure.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.

- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Check Assertion on Unsigned Integer

```
#include <assert.h>

void asserting_x(unsigned int theta) {
    theta =+ 5;
    assert(theta < 0);
}
```

In this example, the `assert` function checks if the input variable, `theta`, is less than or equal to zero. The assertion fails because `theta` is an unsigned integer, so the value at the beginning of the function is at least zero. The += statement increases this positive value by five. Therefore, the range of `theta` is `[5..MAX_INT]`. `theta` is always greater than zero.

#### Correction — Change Assert Expression

One possible correction is to change the assertion expression. By changing the *less-than-or-equal-to* sign to a *greater-than-or-equal-to* sign, the assertion does not fail.

```
#include <assert.h>

void asserting_x(unsigned int theta) {
    theta =+ 5;
    assert(theta > 0);
}
```

#### Correction — Fix Code

One possible correction is to fix the code related to the assertion expression. If the assertion expression is true, fix your code so the assertion passes.

```
#include <assert.h>
#include <stdlib.h>

void asserting_x(int theta) {
    theta = -abs(theta);
    assert(theta < 0);
}
```

#### Example — Asserting Zero

```
#include <assert.h>

#define FLAG 0

int main(void){
    int i_test_z = 0;
    float f_test_z = (float)i_test_z;

    assert(i_test_z);
    assert(f_test_z);
    assert(FLAG);

    return 0;
}
```

In this example, Polyspace does not flag `assert(FLAG)` as a violation because a macro defines `FLAG` as `0`. The Polyspace Bug Finder assertion checker does not flag assertions with a constant zero parameter, `assert(0)`. These types of assertions are commonly used as dynamic checks during runtime. By inserting `assert(0)`, you indicate that the program must not reach this statement during run time, otherwise the program crashes.

However, the assertion checker does flag failed assertions caused by a variable value equal to zero, as seen in the example with `assert(i_test_z)` and `assert(f_test_z)`.

## Check Information

**Category:** Error Conditions, Return Values, Status Codes

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-617

# CWE Rule 676

Use of Potentially Dangerous Function

## Description

### Rule Description

*The program invokes a potentially dangerous function that could introduce a vulnerability if it is used incorrectly, but the function can also be used safely.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Buffer overflow from incorrect string format specifier**
- **Data race through standard library function call**
- **Destination buffer overflow in string manipulation**
- **Unsafe call to a system function**
- **Unsafe conversion from string to numerical value**
- **Use of dangerous standard function**
- **Use of rand() for generating pseudorandom number**

## Examples

### Buffer overflow from incorrect string format specifier

**Issue**

This issue occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

**Risk**

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

**Fix**

Use a format specifier that is compatible with the memory buffer size.

**Example — Memory Buffer Overflow**

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, `buf` can contain 32 `char` elements. Therefore, the format specifier `%33c` causes a buffer overflow.

#### Correction — Use Smaller Precision in Format Specifier

One possible correction is to read a smaller number of elements into the buffer.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

### Data race through standard library function call

#### Issue

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

This issue occurs when:

1. Multiple tasks call the same standard library function.

   For instance, multiple tasks call the `strerror` function.
2. The calls are not protected using a common protection.

   For instance, the calls are not protected by the same critical section.

Functions flagged by this defect are not guaranteed to be reentrant. A function is reentrant if it can be interrupted and safely called again before its previous invocation completes execution. If a function is not reentrant, multiple tasks calling the function without protection can cause concurrency issues. For the list of functions that are flagged, see CON33-C: Avoid race conditions when using library functions.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**. For more information, see “Configuring Polyspace Multitasking Analysis Manually”.

#### Risk

The functions flagged by this defect are nonreentrant because their implementations can use global or static variables. When multiple tasks call the function without protection, the function call from one task can interfere with the call from another task. The two invocations of the function can concurrently access the global or static variables and cause unpredictable results.

The calls can also cause more serious security vulnerabilities, such as abnormal termination, denial-of-service attack, and data integrity violations.

#### Fix

To fix this defect, do one of the following:

- Use a reentrant version of the standard library function if it exists.

  For instance, instead of `strerror()`, use `strerror_r()` or `strerror_s()`. For alternatives to functions flagged by this defect, see the documentation for CON33-C.
- Protect the function calls using common critical sections or temporal exclusion.

See `Critical section details (-critical-section-begin -critical-section-end)` and `Temporally exclusive tasks (-temporal-exclusions-file)`.

To identify existing protections that you can reuse, see the table and graphs associated with the result. The table shows each pair of conflicting calls. The **Access Protections** column shows existing protections on the calls. To see the function call sequence leading to the conflicts, click the icon. For an example, see below.

**Example — Unprotected Call to Standard Library Function from Multiple Tasks**

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

| Option | Specification |
|---|---|
| `Configure multitasking manually` | ☑ |

| Option | Specification | |
|---|---|---|
| `Tasks (-entry-points)` | `task1`<br>`task2`<br>`task3` | |
| `Critical section details (-critical-section-begin -critical-section-end)` | **Starting routine** | **Ending routine** |
| | `begin_critical_section` | `end_critical_section` |

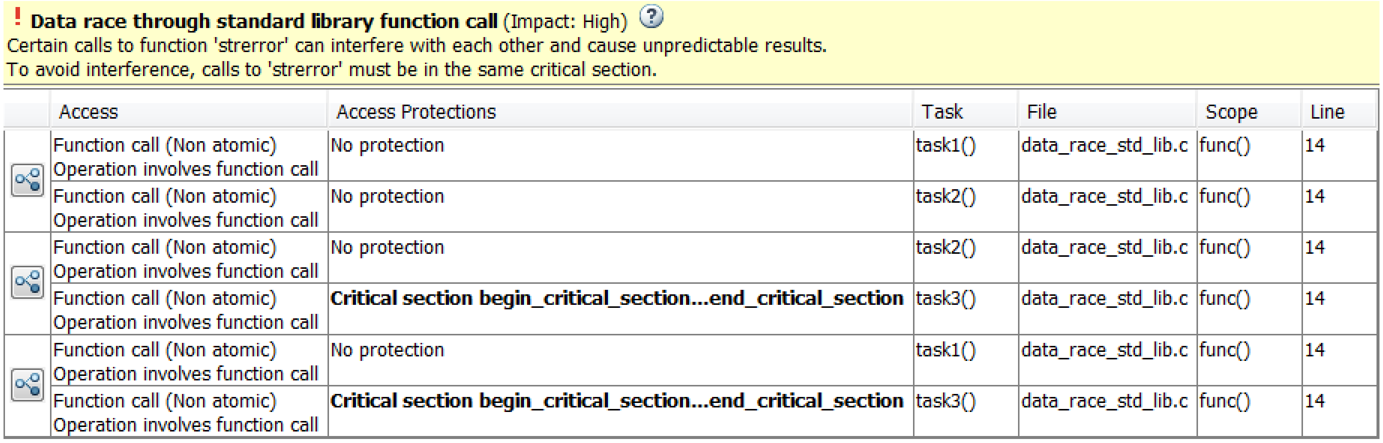
On the command-line, you can use the following:

```
polyspace-bug-finder
  -entry-points task1,task2,task3
  -critical-section-begin begin_critical_section:cs1
  -critical-section-end end_critical_section:cs1
```

In this example, the tasks, `task1`, `task2` and `task3`, call the function `func`. `func` calls the nonreentrant standard library function, `strerror`.

Though `task3` calls `func` inside a critical section, other tasks do not use the same critical section. Operations in the critical section of `task3` are not mutually exclusive with operations in other tasks.

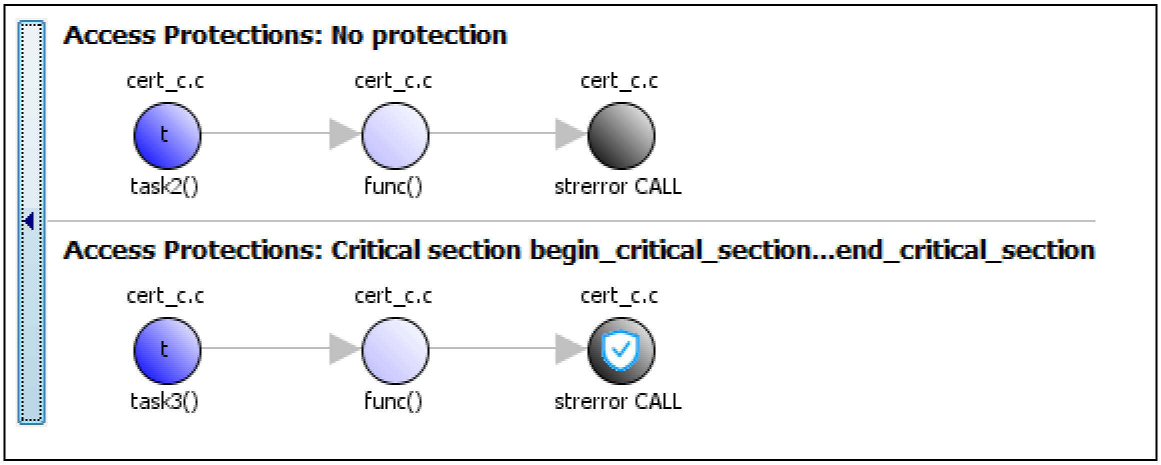
These three tasks are calling a nonreentrant standard library function without common protection. In your result details, you see each pair of conflicting function calls.



! **Data race through standard library function call** (Impact: High)
Certain calls to function 'strerror' can interfere with each other and cause unpredictable results.
To avoid interference, calls to 'strerror' must be in the same critical section.

| | Access | Access Protections | Task | File | Scope | Line |
|---|---|---|---|---|---|---|
| | Function call (Non atomic)<br>Operation involves function call | No protection | task1() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task2() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task2() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | **Critical section begin_critical_section...end_critical_section** | task3() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | No protection | task1() | data_race_std_lib.c | func() | 14 |
| | Function call (Non atomic)<br>Operation involves function call | **Critical section begin_critical_section...end_critical_section** | task3() | data_race_std_lib.c | func() | 14 |

If you click the icon, you see the function call sequence starting from the entry point to the standard library function call. You also see that the call starting from `task3` is in a critical section. The **Access Protections** entry shows the lock and unlock function that begin and end the critical section. In this example, you see the functions `begin_critical_section` and `end_critical_section`.


Access Protections: No protection
cert_c.c
t
task2()
cert_c.c
func()
cert_c.c
strerror CALL
Access Protections: Critical section begin_critical_section...end_critical_section
cert_c.c
t
task3()
cert_c.c
func()
cert_c.c
strerror CALL

**Correction — Use Reentrant Version of Standard Library Function**

One possible correction is to use a reentrant version of the standard library function `strerror`. You can use the POSIX version `strerror_r` which has the same functionality but also guarantees thread-safety.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);
enum { BUFFERSIZE = 64 };

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char errmsg[BUFFERSIZE];
    if (strerror_r(errno, errmsg, BUFFERSIZE) != 0) {
      /* Handle error */
    }
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    func(fptr1);
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     func(fptr2);
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

**Correction — Place Function Call in Critical Section**

One possible correction is to place the call to `strerror` in critical section. You can implement the critical section in multiple ways.

For instance, you can place the call to the intermediate function `func` in the same critical section in the three tasks. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. The calls to `func` and therefore the calls to `strerror` from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call `func` between calls to `begin_critical_section` and `end_critical_section`.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>

void begin_critical_section(void);
void end_critical_section(void);

FILE *getFilePointer(void);

void func(FILE *fp) {
  fpos_t pos;
  errno = 0;
  if (0 != fgetpos(fp, &pos)) {
    char *errmsg = strerror(errno);
    printf("Could not get the file position: %s\n", errmsg);
  }
}

void task1(void) {
    FILE* fptr1 = getFilePointer();
    begin_critical_section();
    func(fptr1);
    end_critical_section();
}

void task2(void) {
     FILE* fptr2 = getFilePointer();
     begin_critical_section();
     func(fptr2);
     end_critical_section();
}

void task3(void) {
     FILE* fptr3 = getFilePointer();
     begin_critical_section();
     func(fptr3);
     end_critical_section();
}
```

#### Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks, `task1`, `task2` and `task3`, temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane, specify the following additional options:

| Option | Value |
|---|---|
| Temporally exclusive tasks (-temporal-exclusions-file) | task1 task2 task3 |

On the command-line, you can use the following:

```
 polyspace-bug-finder
     -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task1 task2 task3
```

### Destination buffer overflow in string manipulation

#### Issue

This issue occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example — Buffer Overflow in sprintf Use

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

**Correction — Use `snprintf` Instead of `sprintf`**

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Unsafe call to a system function

**Issue**

This issue occurs when you use a function that invokes an implementation-defined command processor. These functions include:

- The C standard `system()` function.
- The POSIX `popen()` function.
- The Windows `_popen()` and `_wpopen()` functions.

**Risk**

If the argument of a function that invokes a command processor is not sanitized, it can cause exploitable vulnerabilities. An attacker can execute arbitrary commands or read and modify data anywhere on the system.

**Fix**

Do not use a `system`-family function to invoke a command processor. Instead, use safer functions such as POSIX `execve()` and WinAPI `CreateProcess()`.

**Example — system() Called**

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};

void func(char *arg)
{
    char buf[SIZE512];
    int retval=sprintf(buf, "/usr/bin/any_cmd %s", arg);
    if (retval<=0 || retval>SIZE512){
        /* Handle error */
        abort();
    }
    /* Use of system() to pass any_cmd with
    unsanitized argument to command processor */

    if (system(buf) == -1) {
    /* Handle error */
  }
}
```

In this example, `system()` passes its argument to the host environment for the command processor to execute. This code is vulnerable to an attack by command-injection.

**Correction — Sanitize Argument and Use execve()**

In the following code, the argument of `any_cmd` is sanitized, and then passed to `execve()` for execution. `exec`-family functions are not vulnerable to command-injection attacks.

```
# include <string.h>
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>

enum {
SIZE512=512,
SIZE3=3};


void func(char *arg)
{
  char *const args[SIZE3] = {"any_cmd", arg, NULL};
  char  *const env[] = {NULL};

  /* Sanitize argument */

  /* Use execve() to execute any_cmd. */

  if (execve("/usr/bin/time", args, env) == -1) {
    /* Handle error */
  }
}
```

### Unsafe conversion from string to numerical value

**Issue**

This issue occurs when you perform conversions from strings to integer or floating-point values and your conversion method does not include robust error handling.

**Risk**

Converting a string to numerical value can cause data loss or misinterpretation. Without validation of the conversion or error handling, your program continues with invalid values.

#### Fix

- Add additional checks to validate the numerical value.
- Use a more robust string-to-numeric conversion function such as `strtol`, `strtoll`, `strtoul`, or `strtoull`.

#### Example — Conversion With atoi

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char* argv1)
{
    int s = 0;
    if (demo_check_string_not_empty(argv1))
    {
        s = atoi(argv1);
    }
    return s;
}
```

In this example, `argv1` is converted to an integer with `atoi`. `atoi` does not provide errors for an invalid integer string. The conversion can fail unexpectedly.

#### Correction — Use `strtol` instead

One possible correction is to use `strtol` to validate the input string and the converted integer.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h>

static int demo_check_string_not_empty(char *s)
{
    if (s != NULL)
        return strlen(s) > 0; /* check string null-terminated and not empty */
    else
        return 0;
}

int unsafestrtonumeric(char *argv1)
{
    char *c_str = argv1;
    char *end;
    long sl;

    if (demo_check_string_not_empty(c_str))
    {
        errno = 0; /* set errno for error check */
        sl = strtol(c_str, &end, 10);
        if (end == c_str)
        {
            (void)fprintf(stderr, "%s: not a decimal number\n", c_str);
        }
        else if ('\0' != *end)
        {
            (void)fprintf(stderr, "%s: extra characters: %s\n", c_str, end);
        }
        else if ((LONG_MIN == sl || LONG_MAX == sl) && ERANGE == errno)
        {
            (void)fprintf(stderr, "%s out of range of type long\n", c_str);
        }
        else if (sl > INT_MAX)
        {
            (void)fprintf(stderr, "%ld greater than INT_MAX\n", sl);
        }
        else if (sl < INT_MIN)
        {
            (void)fprintf(stderr, "%ld less than INT_MIN\n", sl);
        }
        else
        {
            return (int)sl;
        }
    }
    return 0;
}
```

### Use of dangerous standard function

#### Issue

This issue occurs when your code uses standard functions that write data to a buffer in a way that can result in buffer overflows.

The following table lists dangerous standard functions, the risks of using each function, and what function to use instead. The checker flags:

- Any use of an inherently dangerous function.
- An use of a possibly dangerous function only if the size of the buffer to which data is written can be determined at compile time. The checker does not flag an use of such a function with a dynamically allocated buffer.

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `gets` | Inherently dangerous — You cannot control the length of input from the console. | `fgets` |

| Dangerous Function | Risk Level | Safer Function |
|---|---|---|
| `std::cin::operator>>` and `std::wcin::operator>>` | Inherently dangerous — You cannot control the length of input from the console. | Preface calls to `cin` by `cin.width` to control the input length. This method can result in truncated input.<br>To avoid potential buffer overflow and truncated input, use `std::string` objects as destinations for >> operator. |
| `strcpy` | Possibly dangerous — If the size of the destination buffer is too small to accommodate the source buffer and a null terminator, a buffer overflow might occur. | Use the function `strlen()` to determine the size of the source buffer, and allocate sufficient memory so that the destination buffer can accommodate the source buffer and a null terminator. Instead of `strcpy`, use the function `strncpy`. |
| `stpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `stpncpy` |
| `lstrcpy` or `StrCpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `StringCbCopy`, `StringCchCopy`, `strncpy`, `strcpy_s`, or `strlcpy` |
| `strcat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `strncat`, `strlcat`, or `strcat_s` |
| `lstrcat` or `StrCat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `StringCbCat`, `StringCchCat`, `strncay`, `strcat_s`, or `strlcat` |
| `wcpcpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcpncpy` |
| `wcscat` | Possibly dangerous — If the concatenated result is greater than the destination, buffer overflow can occur. | `wcsncat`, `wcslcat`, or `wcncat_s` |
| `wcscpy` | Possibly dangerous — If the source length is greater than the destination, buffer overflow can occur. | `wcsncpy` |
| `sprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `snprintf` |
| `vsprintf` | Possibly dangerous — If the output length depends on unknown lengths or values, buffer overflow can occur. | `vsnprintf` |

#### Risk

These functions can cause buffer overflow, which attackers can use to infiltrate your program.

#### Fix

The fix depends on the root cause of the defect. See fixes in the table above and code examples with fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Using sprintf

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (sprintf(dst, "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

This example function uses `sprintf` to copy the string `str` to `dst`. However, if `str` is larger than the buffer, `sprintf` can cause buffer overflow.

#### Correction — Use `snprintf` with Buffer Size

One possible correction is to use `snprintf` instead and specify a buffer size.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

#define BUFF_SIZE 128


int dangerous_func(char *str)
{
    char dst[BUFF_SIZE];
    int r = 0;

    if (snprintf(dst, sizeof(dst), "%s", str) == 1)
    {
        r += 1;
        dst[BUFF_SIZE-1] = '\0';
    }

    return r;
}
```

### Use of rand() for Generating Pseudorandom Number

#### Issue

This issue occurs when you use the function `rand` for generating pseudorandom numbers.

#### Risk

The function `rand` is cryptographically weak. That is, the numbers generated by `rand` can be predictable. Do not use pseudorandom numbers generated from `rand` for security purposes. When a predictable random value controls the execution flow, your program is vulnerable to attacks.

#### Fix

Use more cryptographically sound pseudorandom number generators (PRNG), such as `CryptGenRandom` (Windows), `OpenSSL/RAND_bytes`(Linux/UNIX), or `random` (POSIX).

#### Example — Random Loop Numbers

```
#include <stdio.h>
#include <stdlib.h>

volatile int rd = 1;
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;

    nloops = rand(); //Noncompliant

    for (j = 0; j < nloops; j++) {
        i = rand(); //Noncompliant
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

This example uses `rand` to generate random numbers `nloops` and `i`. The predictability of these variables makes these function vulnerable to attacks.

#### Correction — Use Stronger PRNG

One possible correction is to replace the vulnerable PRNG with a stronger random number generator. For instance, this code uses the PRNG `random()` from POSIX library. `random` is a much stronger PRNG because it can be seeded by a different number every time it is called.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>
#define TIME_UTC 1
volatile int rd = 1;
int randomWrapper(){
    struct timespec ts;
  if (timespec_get(&ts, TIME_UTC) == 0) {
    /* Handle error */
  }
  srandom(ts.tv_nsec ^ ts.tv_sec);  /* Seed the PRNG */
    return random();
}
int main(int argc, char *argv[])
{
    int j, r, nloops;
    struct random_data buf;
    int i = 0;

    nloops = randomWrapper();

    for (j = 0; j < nloops; j++) {
        i = randomWrapper();
        printf("random_r: %ld\n", (long)i);
    }
    return 0;
}
```

### Check Information

**Category:** API / Function Errors

## Version History

**Introduced in R2023a**

### See Also

Check CWE (-cwe)

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

CWE-676

# CWE Rule 685

Function Call With Incorrect Number of Arguments

## Description

### Rule Description

*The software calls a function, procedure, or routine, but the caller specifies too many arguments, or too few arguments, which may lead to undefined behavior and resultant weaknesses.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Format string specifiers and arguments mismatch**
- **Too many va_arg calls for current argument list**
- **Unreliable cast of function pointer**

## Examples

### Format string specifiers and arguments mismatch

#### Issue

This issue occurs when the format specifiers in the formatted output functions such as `printf` do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

#### Risk

Mismatch between format specifiers and the corresponding arguments result in undefined behavior.

#### Fix

Make sure that the format specifiers match the corresponding arguments. For instance, in this example, the `%d` specifier does not match the string argument `message` and the `%s` specifier does not match the integer argument `err_number`.

```
const char *message = "License not available";
int err_number = -4;
printf("Error: %d (error type %s)\n", message, err_number);
```

Switching the two format specifiers fixes the issue. See the specifications for the printf function for more information about format specifiers.

In cases where integer promotion modifies the perceived data type of an argument, the analysis result shows both the original type and the type after promotion. The format specifier has to match the type after integer promotion.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example — Printing a Float

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

#### Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

#### Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
#include <stdio.h>

void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

### Too many va_arg calls for current argument list

#### Issue

This issue occurs when the number of calls to `va_arg` exceeds the number of arguments passed to the corresponding variadic function. The analysis raises a defect only when the variadic function is called.

**Too many va_arg calls for current argument list** does not raise a defect when:

- The number of calls to `va_arg` inside the variadic function is indeterminate. For example, if the calls are from an external source.

- The va_list used in va_arg is invalid.

**Risk**

When you call va_arg and there is no next argument available in va_list, the behavior is undefined. The call to va_arg might corrupt data or return an unexpected result.

**Fix**

Ensure that you pass the correct number of arguments to the variadic function.

**Example — No Argument Available When Calling va_arg**

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
* one named argument 'count'
*/
int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {
/* No further argument available
* in va_list when calling va_arg
*/

            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100);

}
```

In this example, the named argument and only one variadic argument are passed to variadic_func() when it is called inside func(). On the second call to va_arg, no further variadic argument is available in ap and the behavior is undefined.

**Correction — Pass Correct Number of Arguments to Variadic Function**

One possible correction is to ensure that you pass the correct number of arguments to the variadic function.

```
#include <stdarg.h>
#include <stddef.h>
#include <math.h>

/* variadic function defined with
```

```
* one named argument 'count'
*/

int variadic_func(int count, ...) {
    int result = -1;
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        result = va_arg(ap, int);
        count --;
        if (count > 0) {

/* The correct number of arguments is
* passed to va_list when variadic_func()
* is called inside func()
*/
            result += va_arg(ap, int);
        }
    }
    va_end(ap);
    return result;
}

void func(void) {

    (void)variadic_func(2, 100, 200);

}
```

### Unreliable cast of function pointer

#### Issue

This issue occurs when a function pointer is cast to another function pointer that has different argument or return type.

#### Risk

If you cast a function pointer to another function pointer with different argument or return type and then use the latter function pointer to call a function, the behavior is undefined.

#### Fix

Avoid a cast between two function pointers with mismatch in argument or return types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example — Unreliable cast of function pointer error

```
#include <stdio.h>
#include <math.h>
```

```
#define PI 3.142

double Calculate_Sum(int (*fptr)(double))
{
    double  sum = 0.0;
    double  y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;

    fp = sin;
    sum = Calculate_Sum(fp);
    /* Defect: fp implicitly cast to int(*) (double) */

    printf("sum(sin): %f\n", sum);
    return 0;
}
```

The function pointer `fp` is declared as `double (*)(double)`. However in passing it to function `Calculate_Sum`, `fp` is implicitly cast to `int (*)(double)`.

#### Correction — Avoid Function Pointer Cast

One possible correction is to check that the function pointer in the definition of `Calculate_Sum` has the same argument and return type as `fp`. This step makes sure that `fp` is not implicitly cast to a different argument or return type.

```
#include <stdio.h>
#include <math.h>
# define PI 3.142

/*Fix: fptr has same argument and return type everywhere*/
double Calculate_Sum(double (*fptr)(double))
{
    double  sum = 0.0;
    double y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;


    fp = sin;
    sum = Calculate_Sum(fp);
    printf("sum(sin): %f\n", sum);

    return 0;
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

Check CWE (-cwe)

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-685

# CWE Rule 690

Unchecked Return Value to NULL Pointer Dereference

## Description

### Rule Description

*The product does not check for an error after calling a function that can return with a NULL pointer if the function fails, which leads to a resultant NULL pointer dereference.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Returned value of a sensitive function not checked**
- **Tainted NULL or non-null-terminated string**
- **Unprotected dynamic memory allocation**

## Examples

### Returned value of a sensitive function not checked

#### Issue

This issue occurs when you call sensitive standard functions that return information about possible errors and you do one of the following:

- Ignore the return value.

  You simply do not assign the return value to a variable, or explicitly cast the return value to `void`.
- Use an output from the function (return value or argument passed by reference) without testing the return value for errors.

The checker considers a function as sensitive if the function call is prone to failure because of reasons such as:

- Exhausted system resources (for example, when allocating resources).
- Changed privileges or permissions.
- Tainted sources when reading, writing, or converting data from external sources.
- Unsupported features despite an existing API.

The checker only considers functions where the *return value* indicates if the function completed without errors.

Some of these functions can perform critical tasks such as:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)

- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

**Risk**

If you do not check the return value of functions that perform sensitive tasks and indicate error information through their return values, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

**Fix**

Before continuing with the program, test the return value of critical sensitive functions.

For sensitive functions that are not critical, you can explicitly ignore a return value by casting the function to `void`. Polyspace does not raise this defect for sensitive functions cast to void. This resolution is not accepted for *critical sensitive functions* because they perform more vulnerable tasks.

**Example — Sensitive Function Return Ignored**

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

void initialize() {
    pthread_attr_t attr;

    pthread_attr_init(&attr);//Noncompliant
}
int read_file(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r");
  return 0; //Noncompliant

}
```

This example shows calls to the sensitive POSIX functions `pthread_attr_init` and `fmemopen`. Their return values are ignored, causing defect.

**Correction — Cast Function to `(void)`**

One possible correction is to cast the function to void. This fix informs Polyspace and any reviewers that you are explicitly ignoring the return value of the sensitive function.

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>
```

```
void initialize() {
    pthread_attr_t attr;

    (void)pthread_attr_init(&attr);//Compliant
}
int read_file(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  (void)fmemopen (argv[1], strlen (argv[1]), "r"); //Compliant

  return 0;
}
```

**Correction — Test Return Value**

One possible correction is to test the return value of `pthread_attr_init` and `fmemopen` to check for errors.

```
#include <pthread.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

void initialize() {
    pthread_attr_t attr;

    int result = pthread_attr_init(&attr);//Compliant
    if(result != 0){
        //Handle fatal error
    }
}
int read_file(int argc, char *argv[])
{
  FILE *in;
  if (argc != 2) {
    /* Handle error */
  }

  in = fmemopen (argv[1], strlen (argv[1]), "r");
  if (in==NULL){
      // Handle error
  }
  return 0;//Compliant
}
```

**Example — Critical Function Return Ignored**

```
#include <pthread.h>
extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0));
    pthread_join(thread_id,  &res);
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), Polyspace does not ignore this *Return value of a sensitive function not checked* defect. The other critical function, `pthread_join`, returns value that is ignored implicitly. `pthread_join` uses the return value of `pthread_create`, which was not checked.

#### Correction — Test the Return Value of Critical Functions

The correction for this defect is to check the return value of these critical functions to verify the function performed as expected.

```
#include <pthread.h>
#include <stdlib.h>
#define fatal_error() abort()

extern void *start_routine(void *);

void returnnotchecked() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res);
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

### Tainted NULL or non-null-terminated string

#### Issue

This issue occurs when strings from unsecure sources are used in string manipulation routines that implicitly dereference the string buffer, for instance, `strcpy` or `sprintf`.

**Tainted NULL or non-null-terminated string** raises no defect for a string returned from a call to `scanf`-family variadic functions. Similarly, no defect is raised when you pass the string with a `%s` specifier to `printf`-family variadic functions.

#### Risk

If a string is from an unsecure source, it is possible that an attacker manipulated the string or pointed the string pointer to a different memory location.

If the string is NULL, the string routine cannot dereference the string, causing the program to crash. If the string is not null-terminated, the string routine might not know when the string ends. This error can cause you to write out of bounds, causing a buffer overflow.

**Fix**

Validate the string before you use it. Check that:

- The string is not NULL.
- The string is null-terminated
- The size of the string matches the expected size.

**Extend Checker**

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

**Example — Getting String from Input**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));//Noncompliant
    print_str(str);
}
```

In this example, the string `str` is concatenated with the argument `userstr`. The value of `userstr` is unknown. If the size of `userstr` is greater than the space available, the concatenation overflows.

**Correction — Validate the Data**

One possible correction is to check the size of `userstr` and make sure that the string is null-terminated before using it in `strncat`. This example uses a helper function, `sansitize_str`, to validate the string. The defects are concentrated in this function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE128 128
#define MAX 40
extern void print_str(const char*);
int sanitize_str(char* s) {
    int res = 0;
    if (s && (strlen(s) > 0)) { // Noncompliant-TAINTED_STRING only flagged here
        // - string is not null
        // - string has a positive and limited size
        // - TAINTED_STRING on strlen used as a firewall
        res = 1;
    }
    return res;
}
void warningMsg(void)
{
    char userstr[MAX];
    read(0,userstr,MAX);
    char str[SIZE128] = "Warning: ";
    if (sanitize_str(userstr))
        strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}
```

**Correction — Validate the Data**

Another possible correction is to call function `errorMsg` and `warningMsg` with specific strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE128 128

extern void print_str(const char*);

void warningMsg(char* userstr)
{
    char str[SIZE128] = "Warning: ";
    strncat(str, userstr, SIZE128-(strlen(str)+1));
    print_str(str);
}

void errorMsg(char* userstr)
{
  char str[SIZE128] = "Error: ";
  strncat(str, userstr, SIZE128-(strlen(str)+1));
  print_str(str);
}

int manageSensorValue(int sensorValue) {
  int ret = sensorValue;
  if ( sensorValue < 0 ) {
    errorMsg("sensor value should be positive");
    exit(1);
  } else if ( sensorValue > 50 ) {
    warningMsg("sensor value greater than 50 (applying threshold)...");
    sensorValue = 50;
  }

  return sensorValue;
}
```

### Unprotected dynamic memory allocation

#### Issue

This issue occurs when you access dynamically allocated memory without first checking if the prior memory allocation succeeded.

#### Risk

When memory is dynamically allocated using `malloc`, `calloc`, or `realloc`, it returns a value `NULL` if the requested memory is not available. If the code following the allocation accesses the memory block without checking for this `NULL` value, this access is not protected from failures.

#### Fix

Check the return value of `malloc`, `calloc`, or `realloc` for NULL before accessing the allocated memory location.

```
int *ptr = malloc(size * sizeof(int));

if(ptr) /* Check for NULL */
{
   /* Memory access through ptr */
}
```

#### Example — Unprotected dynamic memory allocation error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));

  *p = 2;
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function `calloc` returns `NULL` to `p`. Before accessing the memory through `p`, the code does not check whether `p` is `NULL`

#### Correction — Check for NULL Value

One possible correction is to check whether `p` has value `NULL` before dereference.

```
#include <stdlib.h>

void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

**Example — Unprotected dynamic memory allocation error only on dereference**

```
#include <stdlib.h>
#include<string.h>
typedef struct recordType {
    const char* id;
    const char* data;
} RECORD;

RECORD* MakerecordType(const char *id,unsigned int size){
    RECORD *rec = (RECORD *)calloc(1, sizeof(RECORD));
    rec->id = strdup(id);

    const char *newData = (char *)calloc(1, size);
    rec->data = newData;
    return rec;
}
```

In this example, the checker raises a defect when you dereference the pointer `rec` without checking for a `NULL` value from the prior dynamic memory allocation.

A similar issue happens with the pointer `newData`. However, a defect is not raised because the pointer is not dereferenced but simply copied over to `rec->data`. Simply copying over a possibly null pointer is not an issue by itself. For instance, callers of the `recordType_new` function might check for `NULL` value of `rec->data` before dereferencing, thereby avoiding a null pointer dereference.

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-690

# CWE Rule 704

Incorrect Type Conversion or Cast

## Description

### Rule Description

*The software does not correctly convert an object, resource, or structure from one type to a different type.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Cast to pointer pointing to object of different type**
- **Character value absorbed into EOF**
- **Conversion between pointer and integer**
- **Integer conversion overflow**
- **Misuse of sign-extended character value**
- **Precision loss in integer to float conversion**
- **Qualifier removed in conversion**
- **Reading memory reallocated from object of another type without reinitializing first**
- **Sign change integer conversion overflow**
- **Tainted sign change conversion**
- **Unreliable cast of pointer**
- **Unsafe conversion between pointer and integer**
- **Unsigned integer conversion overflow**
- **Wrong allocated object size for cast**

## Examples

### Cast to pointer pointing to object of different type

#### Issue

The issue occurs when you perform a cast between a pointer to an object type and a pointer to a different object type.

#### Risk

If a pointer to an object is cast into a pointer to a different object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.

Even if the conversion produces a pointer that is correctly aligned, the behavior can be undefined if the pointer is used to access an object.

Exception: You can convert a pointer to object type into a pointer to one of these types:

- `char`
- `signed char`
- `unsigned char`

**Example - Cast to Pointer Pointing to Object of Wider Type**

```
signed   char *p1;
unsigned int *p2;

void foo(void){
  p2 = ( unsigned int * ) p1;     /* Non-compliant */
}
```

In this example, `p1` can point to a `signed char` object. However, `p1` is cast to a pointer that points to an object of wider type, `unsigned int`.

**Example - Cast to Pointer Pointing to Object of Narrower Type**

```
extern unsigned int read_value ( void );
extern void display ( unsigned int n );

void foo ( void ){
  unsigned int u = read_value ( );
  unsigned short *hi_p = ( unsigned short * ) &u;    /* Non-compliant  */
  *hi_p = 0;
  display ( u );
}
```

In this example, `u` is an `unsigned int` variable. `&u` is cast to a pointer that points to an object of narrower type, `unsigned short`.

On a big-endian machine, the statement `*hi_p = 0` attempts to clear the high bits of the memory location that `&u` points to. But, from the result of `display(u)`, you might find that the high bits have not been cleared.

**Example - Cast Adding a Type Qualifier**

```
const short *p;
const volatile short *q;
void foo (void){
  q = ( const volatile short * ) p;  /* Compliant */
}
```

In this example, both `p` and `q` can point to `short` objects. The cast between them adds a `volatile` qualifier only and is therefore compliant.

### Character value absorbed into EOF

**Issue**

This issue occurs when you perform a data type conversion that makes a valid character value indistinguishable from `EOF` (End-of-File). Bug Finder flags the defect in one of the following situations:

- *End-of-File*: You perform a data type conversion such as from `int` to `char` that converts a non-EOF character value into `EOF`.

  ```
  char ch = (char)getchar()
  ```

You then compare the result with EOF.

```
if((int)ch == EOF)
```

The conversion can be explicit or implicit.

- *Wide End-of-File*: You perform a data type conversion that can convert a non-WEOF wide character value into WEOF, and then compare the result with WEOF.

**Risk**

The data type `char` cannot hold the value `EOF` that indicates the end of a file. Functions such as `getchar` have return type `int` to accommodate `EOF`. If you convert from `int` to `char`, the values `UCHAR_MAX` (a valid character value) and `EOF` get converted to the same value -1 and become indistinguishable from each other. When you compare the result of this conversion with `EOF`, the comparison can lead to false detection of `EOF`. This rationale also applies to wide character values and `WEOF`.

**Fix**

Perform the comparison with `EOF` or `WEOF` before conversion.

**Example — Return Value of getchar Converted to char**

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

char func(void)
{
    char ch;
    ch = getchar();
    if (EOF == (int)ch) {
        fatal_error();
    }
    return ch;
}
```

In this example, the return value of `getchar` is implicitly converted to `char`. If `getchar` returns `UCHAR_MAX`, it is converted to -1, which is indistinguishable from EOF. When you compare with EOF later, it can lead to a false positive.

**Correction — Perform Comparison with EOF Before Conversion**

One possible correction is to first perform the comparison with EOF, and then convert from `int` to `char`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

char func(void)
{
    int i;
    i = getchar();
    if (EOF == i) {
        fatal_error();
    }
    else {
        return (char)i;
    }
}
```

### Conversion between pointer and integer

#### Issue

The issue occurs when you convert an integral type into a raw pointer type. Polyspace reports the conversion as a violation of the rule regardless of the sizes of the integral and raw pointer types. For instance, if an integer originates from a pointer and then later, is cast into a pointer, Polyspace flags the conversion from integer to pointer. In this case, the integer and the pointer have the same size.

#### Risk

The mapping between pointers and integers is not always consistent with the addressing structure of the environment.

Converting from integers to pointers can create:

- Misaligned pointers or misaligned objects.
- Invalid pointer addresses.

#### Fix

Where possible, avoid integer-to-pointer conversions. If you want to convert a `void` pointer to an integer, so that you do not change the value, use types:

- C99 — `intptr_t` or `uintptr_t`
- C90 — `size_t` or `ssize_t`

#### Example — Integer to Pointer Conversions

```
unsigned int* badintptrcast(void)
{
    int* ptr;
    unsigned long long int_same_as_ptr ;
    unsigned int int_smaller_than_ptr;

    unsigned int* ptr0 = (unsigned int*)0xdeadbeef; //Noncompliant


    /*int to ptr of same size*/
    ptr = (int*)int_same_as_ptr;  //Noncompliant

    /*int to ptr of different size*/
    ptr = (int*)int_smaller_than_ptr;  //Noncompliant

    return (unsigned int*)(ptr0 - (unsigned int*)ptr);  //Noncompliant
}
```

In this example, Polyspace flags the conversions that might be unsafe. For instance:

- The conversion of `0xdeadbeef` to `unsigned int*` causes alignment issues for the pointer. Polyspace flags the conversion.
- The conversion of the `int_same_as_ptr` and `int_smaller_than_ptr` to `int*` pointers might result in invalid pointer address. Polyspace flags these conversion.

- The `return` statement casts `ptrdiff_t` to a pointer. This pointer might not point to an invalid address. Polyspace flags the conversion.

#### Correction — Use `intptr_t`

One possible correction is to use `intptr_t` types to store addresses. When storing the result of subtracting two pointers, use the type `ptrdiff_t`.

```
#include<stdint.h>
#include<stddef.h>
ptrdiff_t badintptrcast(void)
{
    intptr_t ptr;
    unsigned long long int_same_as_ptr ;
    unsigned int int_smaller_than_ptr;

    intptr_t ptr0 = (intptr_t)0xdeadbeef; //Compliant

    /*int to ptr of same size*/
    ptr = (intptr_t)int_same_as_ptr;  //Compliant

    /*int to ptr of different size*/
    ptr = (intptr_t)int_smaller_than_ptr;  //Compliant

    int offset = 0;
    return (ptrdiff_t)(ptr0 - offset);//Compliant

}
```

### Integer conversion overflow

#### Issue

This issue occurs when converting an integer to a smaller integer type. If the variable does not have enough bytes to represent the original value, the conversion overflows. For instance, if you perform a comparison between implementation-defined type `time_t` and a signed integer, Polyspace reports a violation because `time_t` might be implemented as an unsigned integer.

The checker also detects comparison

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer conversion overflows result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

#### Example — Converting from int to char

```
char convert(void) {

    int num = 1000000;

    return (char)num;
}
```

In the return statement, the integer variable `num` is converted to a `char`. However, an 8-bit or 16-bit character cannot represent 1000000 because it requires at least 20 bits. So the conversion operation overflows.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number.

```
long convert(void) {

    int num = 1000000;

    return (long)num;
}
```

### Misuse of sign-extended character value

#### Issue

This issue occurs when you convert a signed or plain `char` variable containing possible negative values to a wider integer data type (or perform an arithmetic operation that does the conversion) and then use the resulting value in one of these ways:

- For comparison with `EOF` (using == or `!=`)
- As array index

- As argument to a character-handling function in `ctype.h`, for instance, `isalpha()` or `isdigit()`

If you convert a signed `char` variable with a negative value to a wider type such as `int`, the sign bit is preserved (sign extension). This can lead to specific problems even in situations where you think you have accounted for the sign bit.

For instance, the signed `char` value of -1 can represent the character `EOF` (end-of-file), which is an invalid character. Suppose a `char` variable `var` acquires this value. If you treat `var` as a `char` variable, you might want to write special code to account for this invalid character value. However, if you perform an operation such as `var++` (involving integer promotion), it leads to the value 0, which represents a valid value `'\0'` by accident. You transitioned from an invalid to a valid value through the arithmetic operation.

Even for negative values other than -1, a conversion from signed `char` to signed `int` can lead to other issues. For instance, the signed `char` value -126 is equivalent to the `unsigned char` value 130 (corresponding to an extended character `'\202'`). If you convert the value from `char` to `int`, the sign bit is preserved. If you then cast the resulting value to `unsigned int`, you get an unexpectedly large value, 4294967170 (assuming 32-bit `int`). If your code expects the `unsigned char` value of 130 in the final `unsigned int` variable, you can see unexpected results.

The underlying cause of this issue is the sign extension during conversion to a wider type. Most architectures use two's complement representation for storing values. In this representation, the most significant bit indicates the sign of the value. When converted to a wider type, the conversion is done by copying this sign bit to all the leading bits of the wider type, so that the sign is preserved. For instance, the `char` value of -3 is represented as `11111101` (assuming 8-bit `char`). When converted to `int`, the representation is:

```
11111111 11111111 11111111  11111101
```

The value -3 is preserved in the wider type `int`. However, when converted to `unsigned int`, the value (4294967293) is no longer the same as the `unsigned char` equivalent of the original `char` value. If you are not aware of this issue, you can see unexpected results in your code.

**Risk**

In the following cases, Bug Finder flags use of variables after a conversion from `char` to a wider data type or an arithmetic operation that implicitly converts the variable to a wider data type:

- *If you compare the variable value with EOF*:

  A `char` value of -1 can represent the invalid character `EOF` or the valid extended character value `'\377'` (corresponding to the `unsigned char` equivalent, 255). After a `char` variable is cast to a wider type such as `int`, because of sign extension, the `char` value -1, representing one of `EOF` or `'\377'` becomes the `int` value -1, representing only `EOF`. The `unsigned char` value 255 can no longer be recovered from the `int` variable. Bug Finder flags this situation so that you can cast the variable to `unsigned char` first (or avoid the `char`-to-`int` conversion or converting operation before comparison with `EOF`). Only then, a comparison with `EOF` is meaningful. See “Sign-Extended Character Value Compared with EOF” on page 11-125.
- *If you use the variable value as an array index*:

  After a `char` variable is cast to a wider type such as `int`, because of sign extension, all negative values retain their sign. If you use the negative values directly to access an array, you cause buffer overflow/underflow. Even when you account for the negative values, the way you account for them might result in incorrect elements being read from the array. See “Sign-Extended Character Value Used as Array Index” on page 11-126.

- *If you pass the variable value as argument to a character-handling function*:

  According to the C11 standard (Section 7.4), if you supply an integer argument that cannot be represented as unsigned char or EOF, the resulting behavior is undefined. Bug Finder flags this situation because negative char values after conversion can no longer be represented as unsigned char or EOF. For instance, the signed char value -126 is equivalent to the unsigned char value 130, but the signed int value -126 cannot be represented as unsigned char or EOF.

**Fix**

Before conversion to a wider integer data type, cast the signed or plain char value explicitly to unsigned char.

If you use the char data type to not represent characters but simply as a smaller data type to save memory, your use of sign-extended char values might avoid the risks mentioned earlier. If so, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example — Sign-Extended Character Value Compared with EOF**

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = *buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

In this example, the function parser can traverse a string input buf. If a character in the string has the value -1, it can represent either EOF or the valid character value '\377' (corresponding to the unsigned char equivalent 255). When converted to the int variable c, its value becomes the integer value -1, which is always EOF. The later comparison with EOF will not detect if the value returned from parser is actually EOF.

### Correction — Cast to `unsigned char` Before Conversion

One possible correction is to cast the plain `char` value to `unsigned char` before conversion to the wider `int` type. Only then can you test if the return value of `parser` is really `EOF`.

```
#include <stdio.h>
#include <stdlib.h>
#define fatal_error() abort()

extern char parsed_token_buffer[20];

static int parser(char *buf)
{
    int c = EOF;
    if (buf && *buf) {
        c = (unsigned char)*buf++;
    }
    return c;
}

void func()
{
    if (parser(parsed_token_buffer) == EOF) {
        /* Handle error */
        fatal_error();
    }
}
```

### Example — Sign-Extended Character Value Used as Array Index

```
#include <limits.h>
#include <stddef.h>
#include <stdio.h>

#define NUL '\0'
#define SOH 1    /* start of heading */
#define STX 2    /* start of text */
#define ETX 3    /* end of text */
#define EOT 4    /* end of transmission */
#define ENQ 5    /* enquiry */
#define ACK 6    /* acknowledge */

static const int ascii_table[UCHAR_MAX + 1] =
{
      [0]=NUL,[1]=SOH, [2]=STX, [3]=ETX, [4]=EOT, [5]=ENQ,[6]=ACK,
      /* ... */
      [126] = '~',
      /* ... */
      [130/*-126*/]='\202',
      /* ... */
      [255 /*-1*/]='\377'
};

int lookup_ascii_table(char c)
{
    int i;
    i = (c < 0 ? -c : c);
    return ascii_table[i];
}
```

In this example, the `char` variable `c` is converted to the `int` variable `i`. If `c` has negative values, they are converted to positive values before assignment to `i`. However, this conversion can lead to unexpected values when `i` is used as array index. For instance:

- If `c` has the value -1 representing the invalid character `EOF`, you want to probably treat this value separately. However, in this example, a value of `c` equal to -1 leads to a value of `i` equal to 1. The function `lookup_ascii_table` returns the value `ascii_table[1]` (or `SOH`) without the invalid character value `EOF` being accounted for.

  If you use the `char` data type to not represent characters but simply as a smaller data type to save memory, you need not worry about this issue.
- If `c` has a negative value, when assigned to `i`, its sign is reversed. However, if you access the elements of `ascii_table` through `i`, this sign reversal can result in unexpected values being read.

  For instance, if `c` has the value -126, `i` has the value 126. The function `lookup_ascii_table` returns the value `ascii_table[126]` (or `'~'`) but you probably expected the value `ascii_table[130]` (or `'\202'`).

#### Correction – Cast to `unsigned char`

To correct the issues, avoid the conversion from `char` to `int`. First, check `c` for the value `EOF`. Then, cast the value of the `char` variable `c` to `unsigned char` and use the result as array index.

```
#include <limits.h>
#include <stddef.h>
#include <stdio.h>

#define NUL '\0'
#define SOH 1    /* start of heading */
#define STX 2    /* start of text */
#define ETX 3    /* end of text */
#define EOT 4    /* end of transmission */
#define ENQ 5    /* enquiry */
#define ACK 6    /* acknowledge */

static const int ascii_table[UCHAR_MAX + 1] =
{
      [0]=NUL,[1]=SOH, [2]=STX, [3]=ETX, [4]=EOT, [5]=ENQ,[6]=ACK,
      /* ... */
      [126] = '~',
      /* ... */
      [130/*-126*/]='\202',
      /* ... */
      [255 /*-1*/]='\377'
};

int lookup_ascii_table(char c)
{
    int r = EOF;
    if (c != EOF) /* specific handling EOF, invalid character */
        r = ascii_table[(unsigned char)c]; /* cast to 'unsigned char' */
    return r;
}
```

### Precision loss in integer to float conversion

#### Issue

This issue occurs when you cast an integer value to a floating-point type that cannot represent the original integer value.

For instance, the `long int` value `1234567890L` is too large for a variable of type `float` .

#### Risk

If the floating-point type cannot represent the integer value, the behavior is undefined (see C11 standard, 6.3.1.4, paragraph 2). For instance, least significant bits of the variable value can be dropped leading to unexpected results.

#### Fix

Convert to a floating-point type that can represent the integer value.

For instance, if the `float` data type cannot represent the integer value, use the `double` data type instead.

When writing a function that converts an integer to floating point type, before the conversion, check if the integer value can be represented in the floating-point type. For instance, `DBL_MANT_DIG * log2(FLT_RADIX)` represents the number of base-2 digits in the type `double`. Before conversion to the type `double`, check if this number is greater than or equal to the precision of the integer that you are converting. To determine the precision of an integer `num`, use this code:

```
 size_t precision = 0;
 while (num != 0) {
    if (num % 2 == 1) {
      precision++;
    }
    num >>= 1;
 }
```

Some implementations provide a builtin function to determine the precision of an integer. For instance, GCC provides the function `__builtin_popcount`.

#### Example — Conversion of Large Integer to Floating-Point Type

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  float approx = big;
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

In this C code, the `long int` variable `big` is converted to `float`.

#### Correction — Use a Wider Floating-Point Type

One possible correction is to convert to the `double` data type instead of `float`.

```
#include <stdio.h>

int main(void) {
  long int big = 1234567890L;
  double approx = big;
  printf("%ld\n", (big - (long int)approx));
  return 0;
}
```

### Qualifier removed in conversion

#### Issue

*This checker is deactivated in a default Polyspace* as You Code *analysis. See “Checkers Deactivated in Polyspace as You Code Analysis”.*

This issue occurs during a pointer conversion when one pointer has a qualifier and the other does not. For example, when converting from a `const int*` to an `int*`, the conversion removes the `const` qualifier.

This defect applies only for projects in C.

#### Risk

Qualifiers such as `const` or `volatile` in a pointer declaration:

```
const int* ptr;
```

imply that the underlying object is `const` or `volatile`. These qualifiers act as instructions to the compiler. For instance, a `const` object is not supposed to be modified in the code and a `volatile` object is not supposed to be optimized away by the compiler.

If a second pointer points to the same object but does not use the same qualifier, the qualifier on the first pointer is no longer valid. For instance, if a `const int*` pointer and an `int*` pointer point to the same object, you can modify the object through the second pointer and violate the contract implied by the `const` qualifier in the first pointer.

#### Fix

If you intend to convert from one pointer to another, declare both pointers with the same qualifiers.

#### Example — Cast of Character Pointers

```
void read(char *);

void implicit_cast(void) {
    const char  cc, *pcc = &cc;
    char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

During the assignment to the character `q`, the variables, `cc` and `pcc`, are converted from `const char` to `char`. The `const` qualifier is removed during the conversion causing a defect.

#### Correction — Add Qualifiers

One possible correction is to add the same qualifiers to the new variables. In this example, changing `q` to a `const char` fixes the defect.

```
void read(char *);

void implicit_cast(void) {
    const char  cc, *pcc = &cc;
    const char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

#### Correction — Remove Qualifiers

One possible correction is to remove the qualifiers in the converted variable. In this example, removing the `const` qualifier from the `cc` and `pcc` initialization fixes the defect.

```
void read(char *);

void implicit_basic_cast(void) {
    char  cc, *pcc = &cc;
    char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

### Reading memory reallocated from object of another type without reinitializing first

#### Issue

This issue occurs when you do the following in sequence:

1. Reallocate memory to an object with a type that is different from the original allocation.

   For instance, in this code snippet, a memory originally allocated to a pointer with type `struct A*` is reallocated to a pointer with type `struct B*`:

   ```
   struct A;
   struct B;

   struct A *Aptr = (struct A*) malloc(sizeof(struct A));
   struct B *Bptr = (struct B*) realloc(Aptr, sizeof(struct B));
   ```
2. Read from this reallocated memory without reinitializing the memory first.

   Read accesses on the pointer to the reallocated memory can happen through pointer dereference or array indexing. Passing the pointer to a function that takes a pointer to a `const`-qualified object as the corresponding parameter also counts as a read access.

#### Risk

Reading from reallocated memory that has not been reinitialized leads to undefined behavior.

#### Fix

Reinitialize memory after reallocation and before the first read access.

The checker considers any write access on the pointer to the reallocated memory as satisfying the reinitialization requirement (even if the object might only be partially reinitialized). Write accesses on the pointer to the reallocated memory can happen through pointer dereference or array indexing. Passing the pointer to a function that takes a pointer to a non-`const`-qualified object as the corresponding parameter also counts as a write access.

**Example – Noncompliant: Reading from Reallocated Memory Without Reinitializing First**

```
#include<stdlib.h>

struct group {
    char *groupFirst;
    int groupSize;
};

struct groupWithID {
    int groupID;
    char *groupFirst;
    int groupSize;
};

char* readName();
int readSize();

void createGroup(int nextAvailableID) {
    struct group *aGroup;
    struct groupWithID *aGroupWithID;

    aGroup = (struct group*) malloc(sizeof(struct group));

    if(!aGroup) {
        /*Handle error*/
    }

    aGroup->groupFirst = readName();
    aGroup->groupSize  = readSize();

    /* Reassign to group with ID */
    aGroupWithID = (struct groupWithID*) realloc(aGroup, sizeof(struct groupWithID));
    if(!aGroupWithID) {
        free(aGroup);
        /*Handle error*/
    }

    if(aGroupWithID -> groupSize > 0) { /* Noncompliant */
        /* ... */
    }

    /* ...*/
    free(aGroupWithID);
}
```

In this example, the memory allocated to a `group*` pointer using the `malloc` function is reallocated to a `groupWithID*` pointer using the `realloc` function. There is a read access on the reallocated memory before the memory is reinitialized.

#### Correction – Reinitialize Memory After Reallocation and Before First Read

Reinitialize the memory assigned to the `groupWithID*` pointer before the first read access. All bits of the memory can be reinitialized using the `memset` function.

```
#include<stdlib.h>
#include<string.h>

struct group {
    char *groupFirst;
    int groupSize;
};

struct groupWithID {
    int groupID;
    char *groupFirst;
    int groupSize;
};

char* readName();
int readSize();

void createGroup(int nextAvailableID) {
    struct group *aGroup;
    struct groupWithID *aGroupWithID;

    aGroup = (struct group*) malloc(sizeof(struct group));

    if(!aGroup) {
        /*Handle error*/
    }

    aGroup->groupFirst = readName();
    aGroup->groupSize  = readSize();

    /* Reassign to group with ID */
    aGroupWithID = (struct groupWithID*) realloc(aGroup, sizeof(struct groupWithID));
    if(!aGroupWithID) {
        free(aGroup);
        /*Handle error*/
    }

    memset(aGroupWithID, 0 , sizeof(struct groupWithID));
    /* Reinitialize group */
    if(aGroupWithID -> groupSize > 0) {
        /* ... */
    }

    /* ...*/
    free(aGroupWithID);
}
```

### Sign change integer conversion overflow

#### Issue

This issue occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface" or "Interpret Bug Finder Results in Polyspace Access Web Interface".

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

#### Example — Convert from unsigned char to char

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

#### Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

### Tainted sign change conversion

#### Issue

This issue occurs when values from unsecure sources are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
memmove
strncmp
strncpy
calloc
malloc
memalign
```

#### Risk

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

#### Fix

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size); //Noncompliant
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

**Correction — Check Value of size**

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

### Unreliable cast of pointer

**Issue**

This issue occurs when a pointer is implicitly cast to a data type different from its declaration type. Such an implicit casting can take place, for instance, when a pointer to data type `char` is assigned the address of an integer.

This defect applies only if the code language for the project is C.

**Risk**

Casting a pointer to data type different from its declaration type can result in issues such as buffer overflow. If the cast is implicit, it can indicate a coding error.

**Fix**

Avoid *implicit* cast of a pointer to a data type different from its declaration type.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

**Example — Unreliable cast of pointer error**

```
#include <string.h>

void Copy_Integer_To_String()
{
 int src[]={1,2,3,4,5,6,7,8,9,10};
 char buffer[]="Buffer_Text";
 strcpy(buffer,src);
 /* Defect: Implicit cast of (int*) to (char*) */
}
```

`src` is declared as an `int*` pointer. The `strcpy` statement, while copying to `buffer`, implicitly casts `src` to `char*`.

**Correction — Avoid Pointer Cast**

One possible correction is to declare the pointer `src` with the same data type as `buffer`.

```
#include <string.h>
 void Copy_Integer_To_String()
{
 /* Fix: Declare src with same type as buffer */
 char *src[10]={"1","2","3","4","5","6","7","8","9","10"};
 char *buffer[10];

 for(int i=0;i<10;i++)
   buffer[i]="Buffer_Text";

 for(int i=0;i<10;i++)
   buffer[i]= src[i];
 }
```

### Unsafe conversion between pointer and integer

**Issue**

This issue occurs when you convert between a pointer type, such as `intptr_t`, or `uintprt_t`, and an integer type, such as `enum`, `ptrdiff_t`, or `pid_t`, or vice versa.

**Risk**

The mapping between pointers and integers is not always consistent with the addressing structure of the environment.

Converting from pointers to integers can create:

- Truncated or out of range integer values.
- Invalid integer types.

Converting from integers to pointers can create:

- Misaligned pointers or misaligned objects.

- Invalid pointer addresses.

#### Fix

Where possible, avoid pointer-to-integer or integer-to-pointer conversions. If you want to convert a `void` pointer to an integer, so that you do not change the value, use types:

- C99 — `intptr_t` or `uintptr_t`
- C90 — `size_t` or `ssize_t`

#### Example — Integer to Pointer Conversions

```
 unsigned int *badintptrcast(void)
{
    unsigned int *ptr0 = (unsigned int *)0xdeadbeef;
    char *ptr1 = (char *)0xdeadbeef;
    unsigned int *explicit_ptr = reinterpret_cast<unsigned int*>(0xdeadbeef);
    return (unsigned int *)(ptr0 - (unsigned int *)ptr1);
}
```

In this example, there are four conversions, three unsafe conversions and one safe conversion.

- The conversion of `0xdeadbeef` to an `unsigned int*` causes alignment issues for the pointer. Polyspace flags this conversion.
- The conversion of `0xdeadbeef` to a `char *` is not flagged because there are no alignment issues for `char`.
- The explicit `reinterpret_cast` of `0xdeadbeef` to an `unsigned int*` causes alignment issues for the pointer. Polyspace flags this conversion.
- The conversion in the return casts `ptrdiff_t` to a pointer. This pointer might point to an invalid address. Polyspace flags this conversion.

#### Correction — Use `intptr_t`

One possible correction is to use `intptr_t` types to store the pointer address `0xdeadbeef`. Also, you can change the second pointer to an integer offset so that there is no longer a conversion from `ptrdiff_t` to a pointer.

```
#include <stdint.h>

unsigned int *badintptrcast(void)
{
    intptr_t iptr0 = (intptr_t)0xdeadbeef;
    int offset = 0;
    return (unsigned int *)(iptr0 - offset);
}
```

### Unsigned integer conversion overflow

#### Issue

This issue occurs when converting an unsigned integer to a smaller unsigned integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See `Target processor type (-target)`.

#### Risk

Integer conversion overflows result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the conversion so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

In general, avoid conversions to smaller integer types.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Converting from int to char

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum;
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

#### Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned long)unum;
}
```

### Wrong allocated object size for cast

#### Issue

This issue occurs during pointer conversion when the pointer's address is misaligned. If a pointer is converted to a different pointer type, the size of the allocated memory must be a multiple of the size of the destination pointer.

#### Risk

Dereferencing a misaligned pointer has undefined behavior and can cause your program to crash.

#### Fix

Suppose you convert a pointer `ptr1` to `ptr2`. If `ptr1` points to a buffer of `N` bytes and `ptr2` is a *`type`* `*` pointer where `sizeof(`*`type`*`)` is `n` bytes, make sure that `N` is an integer multiple of `n`.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Dynamic Allocation of Pointers

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(13);
    long *dest;

    dest = (long*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to a `long*`. The dynamically allocated memory of `ptr`, 13 bytes, is not a multiple of the size of `dest`, 4 bytes. This misalignment causes the **Wrong allocated object size for cast** defect.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the allocated memory to 12 instead of 13.

```
#include <stdlib.h>

void dyn_non_align(void){
    void *ptr = malloc(12);
    long *dest;
    dest = (long*)ptr;
}
```

#### Example — Static Allocation of Pointers

```
void static_non_align(void){
    char arr[13], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to an `int*` in line 6. `ptr` has a memory size of 13 bytes because the array `arr` has a size of 13 bytes. The size of `dest` is 4 bytes, which is not a multiple of 13. This misalignment causes the **Wrong allocated object size for cast** defect.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the size of the array `arr` to a multiple of 4.

```
void static_non_align(void){
    char arr[12], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr;
}
```

#### Example — Allocation with a Function

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(13);  //defect
    dest2 = (char*)my_alloc(13); //not a defect
}
```

In this example, the software raises a defect on the conversion of the pointer returned by `my_alloc(13)` to an `int*` in line 11. `my_alloc(13)` returns a pointer with a dynamically allocated size of 13 bytes. The size of `dest1` is 4 bytes, which is not a divisor of 13. This misalignment causes the **Wrong allocated object size for cast** defect. In line 12, the same function call, `my_alloc(13)`, does not call a defect for the conversion to `dest2` because the size of `char*`, 1 byte, a divisor of 13.

#### Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the argument for `my_alloc` to a multiple of 4.

```
#include <stdlib.h>

void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(12);
    dest2 = (char*)my_alloc(13);
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-704

# CWE Rule 733

Compiler Optimization Removal or Modification of Security-critical Code

## Description

### Rule Description

*The developer builds a security-critical protection mechanism into the software, but the compiler optimizes the program such that the mechanism is removed or modified.*

### Polyspace Implementation

The rule checker checks for **Uncertain memory cleaning**.

## Examples

### Uncertain memory cleaning

#### Issue

This issue occurs when you clean memory by writing `0` into it but the memory is not read after the cleaning operation. Because the memory is not read after the cleaning, a compiler might skip the cleaning operation when optimizing the code. For instance, consider this code:

```
void foo(){
    void* buffer = getSensitiveData();
    //...
    memset(buffer, 0, sizeof(buffer));
}
```

Here, `buffer` might contain sensitive data. The call to `memset()` is intended to scrub the data. Because buffer is not read after the scrubbing, the compiler optimization process might ignore the `memset()` command. The data in `buffer` might remain uncleaned. Polyspace flags such uncertain memory cleaning. Polyspace does not raise this defect if either of these conditions are true:

- The return value of the memory cleaning function is assigned to a variable.
- The memory cleaning function scrubs a global object.

#### Risk

When a variable is written with no further read, compilers might consider it as a deadstore. To optimize the code, the compiler might remove any assignments to deadstore variables.

If such code optimization removes code that scrubs sensitive information from memory, then sensitive information might remain uncleaned in your system. An attacker can then access the sensitive information and use it to further erode the protection mechanisms in your code.

#### Fix

When scrubbing memory, use code that the compiler cannot remove. For instance, instead of `memset`, use secure functions, such as `memset_s` or `fill_n`. If you use an older version of C or C++ where these functions are unavailable, consider updating your code to a more recent version. Alternatively, you might implement your own secure memory-scrubbing function. When using your own implementation, review the compiled code to make sure that the memory-cleaning operations are untouched by the compiler.

**Example — Avoid Uncertainty in Cleaning Sensitive Memory**

```
#include <string.h>

extern int getSensitiveData(char*, size_t);
extern int writeDB(char*, char*);
char Gbuffer[64];
void Job(char *key) {
    char buffer[64];
    if (getSensitiveData(buffer, sizeof(buffer))) {
        if (writeDB(key, buffer)) {
            // Record data
        }
    }
    (void) memset(buffer, 0, sizeof(buffer)); // Defect
}

void manageGBuffer(char *key) {
    //...
    memset(Gbuffer, 0, sizeof(Gbuffer)); // No Defect
}
```

In this example, the command `memset()` is used in `Job()` to clean the memory in `buffer`. Because `buffer` is not read after the cleaning, the compiler might remove this code when optimizing the code. Polyspace flags the uncertain cleaning. In the function `manage()`, `memset()` is used to clean the global array `Gbuffer`. Polyspace does not flag this cleaning operation.

**Correction — Use Secure Functions to Clean Sensitive Memory**

To fix this defect, use secure functions, such as `memset_s()`, to clean uncertain memory.

```
#include <string.h>

extern int getSensitiveData(char*, size_t);
extern int writeDB(char*, char*);

void Job(char *key) {
    char buffer[64];
    if (getSensitiveData(buffer, sizeof(buffer))) {
        if (writeDB(key, buffer)) {
            // Record data
        }
    }
    memset_s(buffer, 0, sizeof(buffer)); // No Defect
}
```

## Check Information

**Category:** Behavioral Problems

## Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-733

# CWE Rule 762

Mismatched Memory Management Routines

## Description

### Rule Description

*The application attempts to return a memory resource to the system, but it calls a release function that is not compatible with the function that was originally used to allocate that resource.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Invalid free of pointer**
- **Mismatched alloc/dealloc functions on Windows**

## Examples

### Invalid free of pointer

**Issue**

This issue occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

**Risk**

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function or a previous `malloc` function on a different pointer.

**Fix**

In most cases, you can fix the issue by removing the `free` statement. If the pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

**Example — Invalid Free of Pointer Error**

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is deallocated using the `free` function. However, `p` points to a memory location that was not dynamically allocated.

#### Correction — Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

#### Correction — Introduce Pointer Allocation

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p`.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

### Mismatched alloc/dealloc functions on Windows

#### Issue

This issue occurs when you use a Windows deallocation function that is not properly paired to its corresponding allocation function.

#### Risk

Deallocating memory with a function that does not match the allocation function can cause memory corruption or undefined behavior. If you are using an older version of Windows, the improper function can also cause compatibility issues with newer versions.

#### Fix

Properly pair your allocation and deallocation functions according to the functions listed in this table.

| Allocation Function | Deallocation Function |
|---|---|
| malloc() | free() |
| realloc() | free() |
| calloc() | free() |
| _aligned_malloc() | _aligned_free() |
| _aligned_offset_malloc() | _aligned_free() |
| _aligned_realloc() | _aligned_free() |
| _aligned_offset_realloc() | _aligned_free() |
| _aligned_recalloc() | _aligned_free() |
| _aligned_offset_recalloc() | _aligned_free() |
| _malloca() | _freea() |
| LocalAlloc() | LocalFree() |
| LocalReAlloc() | LocalFree() |
| GlobalAlloc() | GlobalFree() |
| GlobalReAlloc() | GlobalFree() |
| VirtualAlloc() | VirtualFree() |
| VirtualAllocEx() | VirtualFreeEx() |
| VirtualAllocExNuma() | VirtualFreeEx() |
| HeapAlloc() | HeapFree() |
| HeapReAlloc() | HeapFree() |

**Example — Memory Deallocated with Incorrect Function**

```
#ifdef _WIN32_
#include <windows.h>
#else
#define _WIN32_
typedef void *HANDLE;
typedef HANDLE HGLOBAL;
typedef HANDLE HLOCAL;
typedef unsigned int UINT;
extern HLOCAL LocalAlloc(UINT uFlags, UINT uBytes);
extern HLOCAL LocalFree(HLOCAL hMem);
extern HGLOBAL GlobalFree(HGLOBAL hMem);
#endif

#define SIZE9 9


void func(void)
{
    /* Memory allocation */
    HLOCAL p = LocalAlloc(0x0000, SIZE9);

    if (p) {
        /* Memory deallocation. */
        GlobalFree(p);
    }
}
```

In this example, memory is allocated with `LocallAlloc()`. The program then erroneously uses `GlobalFree()` to deallocate the memory.

**Correction — Properly Pair Windows Allocation and Deallocation Functions**

When you allocate memory with `LocalAllocate()`, use `LocalFree()` to deallocate the memory.

```
#ifdef _WIN32_
#include <windows.h>
#else
#define _WIN32_
typedef void *HANDLE;
typedef HANDLE HGLOBAL;
typedef HANDLE HLOCAL;
typedef unsigned int UINT;
extern HLOCAL LocalAlloc(UINT uFlags, UINT uBytes);
extern HLOCAL LocalFree(HLOCAL hMem);
extern HGLOBAL GlobalFree(HGLOBAL hMem);
#endif

#define SIZE9 9
void func(void)
{
    /* Memory allocation */
    HLOCAL p = LocalAlloc(0x0000, SIZE9);
    if (p) {
        /* Memory deallocation. */
        LocalFree(p);
    }
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

**Topics**

"Check for and Review Coding Standard Violations"

**External Websites**

CWE-762

# CWE Rule 763

Release of Invalid Pointer or Reference

## Description

### Rule Description

*The application attempts to return a memory resource to the system, but calls the wrong release function or calls the appropriate release function incorrectly.*

### Polyspace Implementation

The rule checker checks for **Invalid free of pointer**.

## Examples

### Invalid free of pointer

#### Issue

This issue occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

#### Risk

The `free` function releases a block of memory allocated on the heap. If you try to access a location on the heap that you did not allocate previously, a segmentation fault can occur.

The issue can highlight coding errors. For instance, you perhaps wanted to use the `free` function or a previous `malloc` function on a different pointer.

#### Fix

In most cases, you can fix the issue by removing the `free` statement. If the pointer is not allocated memory from the heap with `malloc` or `calloc`, you do not need to free the pointer. You can simply reuse the pointer as required.

If the issue highlights a coding error such as use of `free` or `malloc` on the wrong pointer, correct the error.

If the issue occurs because you use the `free` function to free memory allocated with the `new` operator, replace the `free` function with the `delete` operator.

#### Example — Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is deallocated using the `free` function. However, `p` points to a memory location that was not dynamically allocated.

#### Correction — Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

#### Correction — Introduce Pointer Allocation

If the number of elements of the array `p` is not known at compile time, one possible correction is to dynamically allocate memory to the array `p`.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

## Check Information

**Category:** Pointer Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-763

# CWE Rule 766

Critical Data Element Declared Public

## Description

### Rule Description

*The software declares a critical variable, field, or member to be public when intended security policy requires it to be private.*

### Polyspace Implementation

The rule checker checks for **Critical data member is not private**.

## Examples

### Critical data member is not private

#### Issue

This issue occurs when you declare a critical nonstatic data member of a class to be `public`. By default, Polyspace assumes that no data member is critical. Specify the critical data members in your code by using the code behavior `CRITICAL_DATA`. See “Specifying Critical Data Members”. If you do not specify any critical data members, Polyspace raises a warning during the analysis.

#### Risk

Declaring the critical data members as `public` allows the clients of a class to modify critical data members. You can inadvertently introduce vulnerabilities when critical data members are public. The vulnerabilities of such code are difficult to find and time-consuming to fix.

#### Fix

To fix this defect, determine which data members are critical and declare them as `private` or `protected`.

#### Example — Declare Critical Data Members as Private

```
#include <string.h>
#define MAX_PASSWORD_LENGTH 15
#define MAX_USERNAME_LENGTH 15

class UserAccount
{
public:
  UserAccount(char *username, char *password)
  {
    //...
  }

  int authorizeAccess(char *username, char *password)
  {
    //...
  }
  char username[MAX_USERNAME_LENGTH+1];
  char password[MAX_PASSWORD_LENGTH+1];
};
```

In this example, the data members `username` and `password` are declared as `public`. Specify these variables as critical in a code behavior XML file:

```
<specifications>
   <members>
    <member name="password" kind="variable">
        <behavior name="CRITICAL_DATA"/>
    </member>
    <member name="username" kind="variable">
        <behavior name="CRITICAL_DATA"/>
    </member>
   </members>
</specifications>
```

After you specify the variables as critical, Polyspace flags the public critical data members. If you do not specify the critical data members, Polyspace assumes that no data members are critical and the defect is not raised.

#### Correction — Declare Critical Variables as Private

To fix this defect, declare the critical variables as `private`

```
#include <string.h>


#define MAX_PASSWORD_LENGTH 15
#define MAX_USERNAME_LENGTH 15

class UserAccount
{
public:
  UserAccount(char *username, char *password)
  {
    //...
  }

  int authorizeAccess(char *username, char *password)
  {
    //...
  }
private:
  char username[MAX_USERNAME_LENGTH+1];
  char password[MAX_PASSWORD_LENGTH+1];
};
```

## Check Information

**Category:** Permission Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-766

# CWE Rule 767

Access to Critical Private Variable via Public Method

## Description

### Rule Description

*The software defines a public method that reads or modifies a private variable.*

### Polyspace Implementation

The rule checker checks for **Return of non-const handle to encapsulated data member**.

## Examples

### Return of non-const handle to encapsulated data member

#### Issue

This issue occurs when:

- A class method returns a handle to a data member. Handles include pointers and references.
- The method is more accessible than the data member. For instance, the method has access specifier `public`, but the data member is `private` or `protected`.

#### Risk

The access specifier determines the accessibility of a class member. For instance, a class member declared with the `private` access specifier cannot be accessed outside a class. Therefore, nonmember, nonfriend functions cannot modify the member.

When a class method returns a handle to a less accessible data member, the member accessibility changes. For instance, if a `public` method returns a pointer to a `private` data member, the data member is effectively not `private` anymore. A nonmember, nonfriend function calling the `public` method can use the returned pointer to view and modify the data member.

Also, if you assign the pointer to a data member of an object to another pointer, when you delete the object, the second pointer can be left dangling. The second pointer points to the part of an object that does not exist anymore.

#### Fix

One possible fix is to avoid returning a handle to a data member from a class method. Return a data member by value so that a copy of the member is returned. Modifying the copy does not change the data member.

If you must return a handle, use a `const` qualifier with the method return type so that the handle allows viewing, but not modifying, the data member.

#### Example — Return of Pointer to private Data Member

```
#include <string>
#define NUM_RECORDS 100
```

```
struct Date {
    int dd;
    int mm;
    int yyyy;
};


struct Period {
    Date startDate;
    Date endDate;
};

class DataBaseEntry {
private:
    std::string employeeName;
    Period employmentPeriod;
public:
    Period* getPeriod(void);
};

Period* DataBaseEntry::getPeriod(void) {
    return &employmentPeriod;
}


void use(Period*);
void reset(Period*);

int main() {
    DataBaseEntry dataBase[NUM_RECORDS];
    Period* tempPeriod;
    for(int i=0;i < NUM_RECORDS;i++) {
        tempPeriod = dataBase[i].getPeriod();
        use(tempPeriod);
        reset(tempPeriod);
    }
    return 0;
}

void reset(Period* aPeriod) {
       aPeriod->startDate.dd = 1;
       aPeriod->startDate.mm = 1;
       aPeriod->startDate.yyyy = 2000;
}
```

In this example, `employmentPeriod` is `private` to the class `DataBaseEntry`. It is therefore immune from modification by nonmember, nonfriend functions. However, returning a pointer to `employmentPeriod` breaks this encapsulation. For instance, the nonmember function `reset` modifies the member `startDate` of `employmentPeriod`.

**Correction: Return Member by Value**

One possible correction is to return the data member `employmentPeriod` by value instead of pointer. Modifying the return value does not change the data member because the return value is a copy of the data member.

```
#include <string>
#define NUM_RECORDS 100

struct Date {
    int dd;
    int mm;
    int yyyy;
};


struct Period {
    Date startDate;
    Date endDate;
};

class DataBaseEntry {
private:
    std::string employeeName;
    Period employmentPeriod;
public:
    Period getPeriod(void);
};

Period DataBaseEntry::getPeriod(void) {
    return employmentPeriod;
}


void use(Period*);
void reset(Period*);

int main() {
    DataBaseEntry dataBase[NUM_RECORDS];
    Period tempPeriodVal;
    Period* tempPeriod;
    for(int i=0;i < NUM_RECORDS;i++) {
        tempPeriodVal = dataBase[i].getPeriod();
        tempPeriod = &tempPeriodVal;
        use(tempPeriod);
        reset(tempPeriod);
    }
    return 0;
}

void reset(Period* aPeriod) {
       aPeriod->startDate.dd = 1;
       aPeriod->startDate.mm = 1;
       aPeriod->startDate.yyyy = 2000;
}
```

## Check Information

**Category:** Permission Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-767

# CWE Rule 783

Operator Precedence Logic Error

## Description

### Rule Description

*The program uses an expression in which operator precedence causes incorrect logic to be used.*

### Polyspace Implementation

The rule checker checks for **Possibly unintended evaluation of expression because of operator precedence rules**.

## Examples

### Possibly unintended evaluation of expression because of operator precedence rules

#### Issue

This issue occurs when an arithmetic expression result is possibly unintended because operator precedence rules dictate an evaluation order that you do not expect.

The defect highlights expressions of the form `x` *`op_1`* `y` *`op_2`* `z`. Here, *`op_1`* and *`op_2`* are operator combinations that commonly induce this error. For instance, `x == y | z`.

The checker does not flag all operator combinations. For instance, `x == y || z` is not flagged because you most likely intended to perform a logical OR between `x == y` and `z`. Specifically, the checker flags these combinations:

- && and `||`: For instance, `x || y && z` or `x && y || z`.
- Assignment and bitwise operations: For instance, `x = y | z`.
- Assignment and comparison operations: For instance, `x = y != z` or `x = y > z`.
- Comparison operations: For instance, `x > y > z` (except when one of the comparisons is an equality `x == y > z`).
- Shift and numerical operation: For instance, `x << y + 2`.
- Pointer dereference and arithmetic: For instance, `*p++`.

#### Risk

The defect can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:
  - In the operation `*p++`, it is possible that you expect the dereferenced value to be incremented. However, the pointer `p` is incremented before the dereference.
  - In the operation `(x == y | z)`, it is possible that you expect `x` to be compared with `y | z`. However, the == operation happens before the `|` operation.

#### Fix

See if the order of evaluation is what you intend. If not, apply parentheses to implement the evaluation order that you want.

For better readability of your code, it is good practice to apply parenthesis to implement an evaluation order even when operator precedence rules impose that order.

#### Example — Expressions with Possibly Unintended Evaluation Order

```
int test(int a, int b, int c) {
    return(a & b == c);
}
```

In this example, the == operation happens first, followed by the & operation. If you intended the reverse order of operations, the result is not what you expect.

#### Correction — Parenthesis For Intended Order

One possible correction is to apply parenthesis to implement the intended evaluation order.

```
int test(int a, int b, int c) {
    return((a & b) == c);
}
```

## Check Information

**Category:** Behavioral Problems

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-783

# CWE Rule 785

Use of Path Manipulation Function without Maximum-sized Buffer

## Description

### Rule Description

*The software invokes a function for normalizing paths or file names, but it provides an output buffer that is smaller than the maximum possible size, such as PATH_MAX.*

### Polyspace Implementation

The rule checker checks for **Use of path manipulation function without maximum-sized buffer checking**.

## Examples

### Use of path manipulation function without maximum-sized buffer checking

**Issue**

This issue occurs when the destination argument of a path manipulation function such as `realpath` or `getwd` has a buffer size less than `PATH_MAX` bytes.

**Risk**

A buffer smaller than `PATH_MAX` bytes can overflow but you cannot test the function return value to determine if an overflow occurred. If an overflow occurs, following the function call, the content of the buffer is undefined.

For instance, `char *getwd(char *buf)` copies an absolute path name of the current folder to its argument. If the length of the absolute path name is greater than `PATH_MAX` bytes, `getwd` returns `NULL` and the content of `*buf` is undefined. You can test the return value of `getwd` for `NULL` to see if the function call succeeded.

However, if the allowed buffer for `buf` is less than `PATH_MAX` bytes, a failure can occur for a smaller absolute path name. In this case, `getwd` does not return `NULL` even though a failure occurred. Therefore, the allowed buffer for `buf` must be `PATH_MAX` bytes long.

**Fix**

Possible fixes are:

- Use a buffer size of `PATH_MAX` bytes. If you obtain the buffer from an unknown source, before using the buffer as argument of `getwd` or `realpath` function, make sure that the size is less than `PATH_MAX` bytes.
- Use a path manipulation function that allows you to specify a buffer size.

  For instance, if you are using `getwd` to get the absolute path name of the current folder, use `char *getcwd(char *buf, size_t size);` instead. The additional argument `size` allows you to specify a size greater than or equal to `PATH_MAX`.

- Allow the function to allocate additional memory dynamically, if possible.

  For instance, `char *realpath(const char *path, char *resolved_path);` dynamically allocates memory if `resolved_path` is `NULL`. However, you have to deallocate this memory later using the `free` function.

**Example — Possible Buffer Overflow in Use of getwd Function**

```
#include <unistd.h>
#include <linux/limits.h>
#include <stdio.h>

void func(void) {
    char buf[PATH_MAX];
    if (getwd(buf+1)!= NULL)         {
        printf("cwd is %s\n", buf);
    }
}
```

In this example, although the array `buf` has `PATH_MAX` bytes, the argument of `getwd` is `buf + 1`, whose allowed buffer is less than `PATH_MAX` bytes.

**Correction — Use Array of Size PATH_MAX Bytes**

One possible correction is to use an array argument with size equal to `PATH_MAX` bytes.

```
#include <unistd.h>
#include <linux/limits.h>
#include <stdio.h>

void func(void) {
    char buf[PATH_MAX];
    if (getwd(buf)!= NULL)         {
        printf("cwd is %s\n", buf);
    }
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-785

# CWE Rule 787

Out-of-bounds Write

## Description

### Rule Description

*The software writes data past the end, or before the beginning, of the intended buffer.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Destination buffer overflow in string manipulation**
- **Destination buffer underflow in string manipulation**

## Examples

### Destination buffer overflow in string manipulation

#### Issue

This issue occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example — Buffer Overflow in sprintf Use

```
#include <stdio.h>
```

```
void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

**Correction — Use `snprintf` Instead of `sprintf`**

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Destination buffer underflow in string manipulation

**Issue**

This issue occurs when certain string manipulation functions write to their destination buffer argument at a negative offset from the beginning of the buffer.

For instance, for the function `sprintf(char* buffer, const char* format)`, you obtain the `buffer` from an operation `buffer = (char*)arr; ... buffer += offset;`. `arr` is an array and `offset` is a negative value.

**Risk**

Buffer underflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer underflow also introduces the risk of code injection.

**Fix**

If the destination buffer argument results from pointer arithmetic, see if you are decrementing a pointer. Fix the pointer decrement by modifying either the original value before decrement or the decrement value.

**Example — Buffer Underflow in sprintf Use**

```
#include <stdio.h>
#define offset -2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

In this example, `&buffer[offset]` is at a negative offset from the memory allocated to `buffer`.

#### Correction — Change Pointer Decrementer

One possible correction is to change the value of `offset`.

```
#include <stdio.h>
#define offset 2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

## Check Information

**Category:** Memory Buffer Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-787

# CWE Rule 789

Memory Allocation with Excessive Size Value

## Description

### Rule Description

*The product allocates memory based on an untrusted, large size value, but it does not ensure that the size is within expected limits, allowing arbitrary amounts of memory to be allocated.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Memory allocation with tainted size**
- **Tainted size of variable length array**
- **Wrong type used in sizeof**

## Examples

### Memory allocation with tainted size

#### Issue

This issue occurs when a memory allocation function, such as `calloc` or `malloc`, uses a size argument from an unsecure source.

#### Risk

Uncontrolled memory allocation can cause your program to request too much system memory. This consequence can lead to a crash due to an out-of-memory condition, or assigning too many resources.

#### Fix

Before allocating memory, check the value of your arguments to check that they do not exceed the bounds.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Allocate Memory Using Input From User

```
#include<stdio.h>
#include <stdlib.h>

int* bug_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = (int*)malloc(size);//Noncompliant
    return p;
}
```

In this example, `malloc` allocates `size` bytes of memory for the pointer `p`. The variable `size` comes from the user of the program. Its value is not checked, and it could be larger than the amount of available memory. If `size` is larger than the number of available bytes, your program could crash.

#### Correction — Check Size of Memory to be Allocated

One possible correction is to check the size of the memory that you want to allocate before performing the `malloc` operation. This example checks to see if `size` is positive and less than the maximum size.

```
#include<stdio.h>
#include <stdlib.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

int* corrected_taintedmemoryallocsize(void) {
    size_t size;
    scanf("%zu", &size);
    int* p = NULL;
    if (size>0 && size<SIZE128) {          /* Fix: Check entry range before use */
        p = (int*)malloc((unsigned int)size);
    }
    return p;
}
```

### Tainted size of variable length array

#### Issue

This issue occurs when the size of a variable length array (VLA) is obtained from an unsecure source.

#### Risk

If an attacker changed the size of your VLA to an unexpected value, it can cause your program to crash or behave unexpectedly.

If the size is non-positive, the behavior of the VLA is undefined. Your program does not perform as expected.

If the size is unbounded, the VLA can cause memory exhaustion or stack overflow.

#### Fix

Validate your VLA size to make sure that it is positive and less than a maximum value.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

### Example — Example — User Input Argument Used as Size of VLA

```
#include<stdio.h>
#inclule<stdlib.h>
#define LIM 40

long squaredSum(int size) {

    int tabvla[size];
    long res = 0;
    for (int i=0 ; i<LIM-1 ; ++i) {
        tabvla[i] = i*i;
        res += tabvla[i];
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

In this example, a variable length array size is based on an input argument. Because this input argument value is not checked, the size may be negative or too large.

### Correction — Check VLA Size

One possible correction is to check the size variable before creating the variable length array. This example checks if the size is larger than 0 and less than 40, before creating the VLA

```
#include <stdio.h>
#include <stdlib.h>
#define LIM 40

long squaredSum(int size) {
    long res = 0;
    if (size>0 && size<LIM){
        int tabvla[size];
        for (int i=0 ; i<size || i<LIM-1 ; ++i) {
            tabvla[i] = i*i;
            res += tabvla[i];
        }
    }else{
        res = -1;
    }
    return res;
}
int main(){
    int size;
    scanf("%d",&size);
    //...
    long result = squaredSum(size);
    //...
    return 0;
}
```

## Wrong type used in sizeof

### Issue

This issue occurs when both of the following conditions hold:

1. You assign the address of a block of memory to a pointer, or transfer data between two blocks of memory. The assignment or copy uses the `sizeof` operator.

   For instance, you initialize a pointer using `malloc(sizeof(`*`type`*`))` or copy data between two addresses using `memcpy(`*`destination_ptr`*`, `*`source_ptr`*`, sizeof(`*`type`*`))`.
2. You use an incorrect type as argument of the `sizeof` operator. For instance:

   - You might be using the pointer type instead of the type that the pointer points to. For example, to initialize a *`type`*`*` pointer, you might be using `malloc(sizeof(`*`type`*`*))` instead of `malloc(sizeof(`*`type`*`))`.
   - You might be using a completely unrelated type as `sizeof` argument. For example, to initialize a *`type`*`*` pointer, you might be using `malloc(sizeof(`*`anotherType`*`))`.

### Risk

Irrespective of what *`type`* stands for, the expression `sizeof(`*`type`*`*)` always returns a fixed size. The size returned is the pointer size on your platform in bytes. The appearance of `sizeof(`*`type`*`*)` often indicates an unintended usage. The error can cause allocation of a memory block that is much smaller than what you need and lead to weaknesses such as buffer overflows.

For instance, assume that `structType` is a structure with ten `int` variables. If you initialize a `structType*` pointer using `malloc(sizeof(structType*))` on a 32-bit platform, the pointer is assigned a memory block of four bytes. However, to be allocated completely for one `structType` variable, the `structType*` pointer must point to a memory block of `sizeof(structType) = 10 * sizeof(int)` bytes. The required size is much greater than the actual allocated size of four bytes.

### Fix

To initialize a *`type`*`*` pointer, replace `sizeof(`*`type`*`*)` in your pointer initialization expression with `sizeof(`*`type`*`)`.

### Example — Allocate a Char Array With sizeof

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char*) * 5);
    free(str);

}
```

In this example, memory is allocated for the character pointer `str` using a `malloc` of five char pointers. However, `str` is a pointer to a character, not a pointer to a character pointer. Therefore the `sizeof` argument, `char*`, is incorrect.

### Correction — Match Pointer Type to `sizeof` Argument

One possible correction is to match the argument to the pointer type. In this example, `str` is a character pointer, therefore the argument must also be a character.

```
#include <stdlib.h>

void test_case_1(void) {
    char* str;

    str = (char*)malloc(sizeof(char) * 5);
    free(str);

}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-789

# CWE Rule 798

Use of Hard-coded Credentials

## Description

### Rule Description

*The software contains hard-coded credentials, such as a password or cryptographic key, which it uses for its own inbound authentication, outbound communication to external components, or encryption of internal data.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Constant cipher key**
- **Hard-coded sensitive data**

## Examples

### Constant cipher key

**Issue**

This issue occurs when you use a constant for the encryption or decryption key.

**Risk**

If you use a constant for the encryption or decryption key, an attacker can retrieve your key easily.

You use a key to encrypt and later decrypt your data. If a key is easily retrieved, data encrypted using that key is not secure.

**Fix**

Produce a random key by using a strong random number generator.

For a list of random number generators that are cryptographically weak, see `Vulnerable pseudo-random number generator`.

**Example — Constants Used for Key**

```
#include <openssl/evp.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16] = {'1', '2', '3', '4','5','6','b','8','9',
                                 '1','2','3','4','5','6','7'};
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

In this example, the cipher key, key, has constants only. An attacker can easily retrieve a constant key.

#### Correction — Use Random Key

Use a strong random number generator to produce the cipher key. The corrected code here uses the function RAND_bytes declared in openssl/rand.h.

```
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <stdlib.h>
#define SIZE16 16

int func(EVP_CIPHER_CTX *ctx, unsigned char *iv){
    unsigned char key[SIZE16];
    RAND_bytes(key, 16);
    return EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
}
```

### Hard-coded sensitive data

#### Issue

This issue occurs when data that is potentially sensitive is directly exposed in the code, for instance, as string literals. The checker identifies certain data as sensitive from their use in certain functions such as password encryption functions.

Following data can be potentially sensitive.

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Host name | • sethostname, setdomainname, gethostbyname, gethostbyname2, getaddrinfo, gethostbyname_r, gethostbyname2_r (string argument)<br>• inet_aton, inet_pton, inet_net_pton, inet_addr, inet_network (string argument)<br>• mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (2nd argument) |
| Password | • CreateProcessWithLogonW, LogonUser (1st argument)<br>• mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (3rd argument) |

| Type of Data | Functions That Indicate Sensitive Nature of Information |
|---|---|
| Database | • MySQL: mysql_real_connect, mysql_real_connect_nonblocking, mysql_connect (4th argument)<br>• SQLite: sqlite3_open, sqlite3_open16, sqlite3_open_v2 (1st argument)<br>• PostgreSQL: PQconnectdb<br>• Microsoft SQL: SQLDriverConnect (3rd argument) |
| User name | • getpw, getpwnam, getpwnam_r, getpwuid, getpwuid_r |
| Salt | crypt, crypt_r (2nd argument) |
| Cryptography keys and initialization vectors | OpenSSL:<br>• EVP_CipherInit, EVP_EncryptInit, EVP_DecryptInit (3rd argument)<br>• EVP_CipherInit_ex, EVP_EncryptInit_ex, EVP_DecryptInit_ex (4th argument) |
| Seed | • srand, srandom, initstate (1st argument)<br>• OpenSSL: RAND_seed, RAND_add |

**Risk**

Information that is hardcoded can be queried from binaries generated from the code.

**Fix**

Avoid hard coding sensitive information.

**Example — Sensitive Data Exposed Through String Literals**

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"
const char *host = "localhost";
char *user = "guest";
char *passwd;

DbHandle connect_to_database_server(const char *db)
```

```
{
    passwd = (char*)"guest";
    return (DbHandle)
        mysql_real_connect (sql, host, user, passwd, db, 0, 0x0, 0);
}
```

In this example, the arguments `host` (host name), `user` (user name), and `passwd` (password) are string literals and directly exposed in the code.

Querying the generated binary for ASCII strings can reveal this information.

**Correction – Read Sensitive Data from Secured Configuration Files**

One possible correction is to read the data from a configuration file. In the following corrected example, the call to function `connect_to_database_server_init` presumably reads the host name, user name, and password into its arguments from a secured configuration file.

```
// Typically, you include the header "mysql.h" with function and type declarations.
// In this example, only the required lines from the header are quoted.

typedef struct _MYSQL MYSQL;

MYSQL *mysql_real_connect(MYSQL *mysql,
                          const char *host, const char *user, const char *passwd,
                          const char *db, unsigned int port, const char *unix_socket,
                          unsigned long client_flag);

typedef void * DbHandle;
extern MYSQL *sql;

// File that uses functions from "mysql.h"

int connect_to_database_server_init(const char **host,
                                    const char **user,
                                    const char **passwd,
                                    const char **db);

DbHandle connect_to_database_server(const char *db)
{
    const char *host_from_cfg;
    const char *user_from_cfg;
    const char *passwd_from_cfg;
    const char *db_from_cfg;
    if (connect_to_database_server_init(&host_from_cfg,
                                        &user_from_cfg,
                                        &passwd_from_cfg,
                                        &db_from_cfg))
    {
        return (DbHandle)
            mysql_real_connect (sql, host_from_cfg, user_from_cfg,
                        passwd_from_cfg, db_from_cfg,  0, 0x0, 0);
    }
    else
        return (DbHandle)0x0;
}
```

## Check Information

**Category:** Credentials Management Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-798

# CWE Rule 805

Buffer Access with Incorrect Length Value

## Description

### Rule Description

*The software uses a sequential operation to read or write a buffer, but it uses an incorrect length value that causes it to access memory that is outside of the bounds of the buffer.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Buffer overflow from incorrect string format specifier**
- **Destination buffer overflow in string manipulation**
- **Destination buffer underflow in string manipulation**
- **Invalid use of standard library memory routine**
- **Invalid use of standard library routine**
- **Invalid use of standard library string routine**
- **Mismatch between data length and size**
- **Possible misuse of sizeof**

## Examples

### Buffer overflow from incorrect string format specifier

**Issue**

This issue occurs when the format specifier argument for functions such as `sscanf` leads to an overflow or underflow in the memory buffer argument.

**Risk**

If the format specifier specifies a precision that is greater than the memory buffer size, an overflow occurs. Overflows can cause unexpected behavior such as memory corruption.

**Fix**

Use a format specifier that is compatible with the memory buffer size.

**Example — Memory Buffer Overflow**

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%33c", buf);
}
```

In this example, buf can contain 32 char elements. Therefore, the format specifier %33c causes a buffer overflow.

**Correction — Use Smaller Precision in Format Specifier**

One possible correction is to read a smaller number of elements into the buffer.

```
#include <stdio.h>

void func (char *str[]) {
    char buf[32];
    sscanf(str[1], "%32c", buf);
}
```

### Destination buffer overflow in string manipulation

**Issue**

This issue occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function sprintf(char* buffer, const char* format), you use a constant string format of greater size than buffer.

**Risk**

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

**Fix**

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use sprintf to write formatted data to a string, use snprintf, _snprintf or sprintf_s instead to enforce length control. Alternatively, use asprintf to automatically allocate the memory required for the destination buffer.
- If you use vsprintf to write formatted data from a variable argument list to a string, use vsnprintf or vsprintf_s instead to enforce length control.
- If you use wcscpy to copy a wide string, use wcsncpy, wcslcpy, or wcscpy_s instead to enforce length control.

Another possible solution is to increase the buffer size.

**Example — Buffer Overflow in sprintf Use**

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, buffer can contain 20 char elements but fmt_string has a greater size.

#### Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Destination buffer underflow in string manipulation

#### Issue

This issue occurs when certain string manipulation functions write to their destination buffer argument at a negative offset from the beginning of the buffer.

For instance, for the function `sprintf(char* buffer, const char* format)`, you obtain the `buffer` from an operation `buffer = (char*)arr; ... buffer += offset;`. `arr` is an array and `offset` is a negative value.

#### Risk

Buffer underflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer underflow also introduces the risk of code injection.

#### Fix

If the destination buffer argument results from pointer arithmetic, see if you are decrementing a pointer. Fix the pointer decrement by modifying either the original value before decrement or the decrement value.

#### Example — Buffer Underflow in sprintf Use

```
#include <stdio.h>
#define offset -2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";

    sprintf(&buffer[offset], fmt_string);
}
```

In this example, `&buffer[offset]` is at a negative offset from the memory allocated to `buffer`.

#### Correction — Change Pointer Decrementer

One possible correction is to change the value of `offset`.

```
#include <stdio.h>
#define offset 2

void func(void) {
    char buffer[20];
    char *fmt_string ="Text";
    sprintf(&buffer[offset], fmt_string);
}
```

### Invalid use of standard library memory routine

#### Issue

This issue occurs when a memory library function is called with invalid arguments. For instance, the `memcpy` function copies to an array that cannot accommodate the number of bytes copied.

#### Risk

Use of a memory library function with invalid arguments can result in issues such as buffer overflow.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Example — Invalid Use of Standard Library Memory Routine Error

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  char str1[10],str2[5];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

#### Correction — Call Function with Valid Arguments

One possible correction is to adjust the size of str2 so that it accommodates the characters copied with the memcpy function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

### Invalid use of standard library routine

#### Issue

This issue occurs when you use invalid arguments with a function from the standard library. This defect picks up errors related to other functions not covered by float, integer, memory, or string standard library routines.

#### Risk

Invalid arguments to a standard library function result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. For instance, the argument to a printf function can be NULL because a pointer was initialized with NULL and the initialization value was not overwritten along a specific execution path.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Calling printf Without a String

```
#include <stdio.h>
#include <stdlib.h>

void print_null(void) {

  printf(NULL);
}
```

The function printf takes only string input arguments or format specifiers. In this function, the input value is NULL, which is not a valid string.

#### Correction — Use Compatible Input Arguments

One possible correction is to change the input arguments to fit the requirements of the standard library routine. In this example, the input argument was changed to a character.

```
#include <stdio.h>

void print_null(void) {
    char zero_val = '0';
    printf((const char*)zero_val);
}
```

### Invalid use of standard library string routine

#### Issue

This issue occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the strcpy function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the strcpy function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to strcpy. In some cases, you can use an alternative function to avoid the error. For instance, instead of strcpy, you can use strncpy to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";
  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

**Correction — Use Valid Arguments**

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

### Mismatch between data length and size

**Issue**

This issue occurs when you do not check the length argument and data buffer argument of memory copying functions such as `memcpy`, `memset`, or `memmove`, to protect against buffer overflows.

**Risk**

If an attacker can manipulate the data buffer or length argument, the attacker can cause buffer overflow by making the actual data size smaller than the length.

This mismatch in length allows the attacker to copy memory past the data buffer to a new location. If the extra memory contains sensitive information, the attacker can now access that data.

This defect is similar to the SSL Heartbleed bug.

**Fix**

When copying or manipulating memory, compute the length argument directly from the data so that the sizes match.

**Example — Copy Buffer of Data**

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;
```

```
extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    memcpy(&(beta.data[num]), os->data + 2, length);

    return(1);
}
```

This function copies the buffer `alpha` into a buffer `beta`. However, the `length` variable is not related to `data+2`.

#### Correction — Check Buffer Length

One possible correction is to check the length of your buffer against the maximum value minus 2. This check ensures that you have enough space to copy the data to the `beta` structure.

```
#include <stdlib.h>
#include <string.h>

typedef struct buf_mem_st {
    char *data;
    size_t max;     /* size of buffer */
} BUF_MEM;

extern BUF_MEM beta;

int cpy_data(BUF_MEM *alpha)
{
    BUF_MEM *os = alpha;
    int num, length;

    if (alpha == 0x0) return 0;
    num = 0;

    length = *(unsigned short *)os->data;
    if (length<(os->max -2)) {
        memcpy(&(beta.data[num]), os->data + 2, length);
    }

    return(1);

}
```

### Possible misuse of sizeof

#### Issue

This issue occurs when Polyspace Bug Finder detects possibly unintended results from the use of `sizeof` operator. For instance:

- You use the sizeof operator on an array parameter name, expecting the array size. However, the array parameter name by itself is a pointer. The sizeof operator returns the size of that pointer.
- You use the sizeof operator on an array element, expecting the array size. However, the operator returns the size of the array element.
- The size argument of certain functions such as strncmp or wcsncpy is incorrect because you used the sizeof operator earlier with possibly incorrect expectations. For instance:
  - In a function call strncmp(string1, string2, num), num is obtained from an incorrect use of the sizeof operator on a pointer.
  - In a function call wcsncpy(destination, source, num), num is the not the number of wide characters but a size in bytes obtained by using the sizeof operator. For instance, you use wcsncpy(destination, source, sizeof(destination) - 1) instead of wcsncpy(destination, source, (sizeof(desintation)/sizeof(wchar_t)) - 1).

**Risk**

Incorrect use of the sizeof operator can cause the following issues:

- If you expect the sizeof operator to return array size and use the return value to constrain a loop, the number of loop runs are smaller than what you expect.
- If you use the return value of sizeof operator to allocate a buffer, the buffer size is smaller than what you require. Insufficient buffer can lead to resultant weaknesses such as buffer overflows.
- If you use the return value of sizeof operator incorrectly in a function call, the function does not behave as you expect.

**Fix**

Possible fixes are:

- Do not use the sizeof operator on an array parameter name or array element to determine array size.

  The best practice is to pass the array size as a separate function parameter and use that parameter in the function body.
- Use the sizeof operator carefully to determine the number argument of functions such as strncmp or wcsncpy. For instance, for wide string functions such as wcsncpy, use the number of wide characters as argument instead of the number of bytes.

**Example — sizeof Used Incorrectly to Determine Array Size**

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < sizeof(a)/sizeof(int); i++)    {
        a[i] = i + 1;
    }
}
```

In this example, sizeof(a) returns the size of the pointer a and not the array size.

**Correction — Determine Array Size in Another Way**

One possible correction is to use another means to determine the array size.

```
#define MAX_SIZE 1024

void func(int a[MAX_SIZE]) {
    int i;

    for (i = 0; i < MAX_SIZE; i++)    {
        a[i] = i + 1;
    }
}
```

## Check Information

**Category:** Memory Buffer Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-805

# CWE Rule 806

Buffer Access Using Size of Source Buffer

## Description

### Rule Description

*The software uses the size of a source buffer when reading from or writing to a destination buffer, which may cause it to access memory that is outside of the bounds of the buffer.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Destination buffer overflow in string manipulation**
- **Invalid use of standard library string routine**

## Examples

### Destination buffer overflow in string manipulation

#### Issue

This issue occurs when certain string manipulation functions write to their destination buffer argument at an offset greater than the buffer size.

For instance, when calling the function `sprintf(char* buffer, const char* format)`, you use a constant string `format` of greater size than `buffer`.

#### Risk

Buffer overflow can cause unexpected behavior such as memory corruption or stopping your system. Buffer overflow also introduces the risk of code injection.

#### Fix

One possible solution is to use alternative functions to constrain the number of characters written. For instance:

- If you use `sprintf` to write formatted data to a string, use `snprintf`, `_snprintf` or `sprintf_s` instead to enforce length control. Alternatively, use `asprintf` to automatically allocate the memory required for the destination buffer.
- If you use `vsprintf` to write formatted data from a variable argument list to a string, use `vsnprintf` or `vsprintf_s` instead to enforce length control.
- If you use `wcscpy` to copy a wide string, use `wcsncpy`, `wcslcpy`, or `wcscpy_s` instead to enforce length control.

Another possible solution is to increase the buffer size.

#### Example — Buffer Overflow in sprintf Use

```
#include <stdio.h>
```

```
void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    sprintf(buffer, fmt_string);
}
```

In this example, `buffer` can contain 20 `char` elements but `fmt_string` has a greater size.

#### Correction — Use `snprintf` Instead of `sprintf`

One possible correction is to use the `snprintf` function to enforce length control.

```
#include <stdio.h>

void func(void) {
    char buffer[20];
    char *fmt_string = "This is a very long string, it does not fit in the buffer";

    snprintf(buffer, 20, fmt_string);
}
```

### Invalid use of standard library string routine

#### Issue

This issue occurs when a string library function is called with invalid arguments.

#### Risk

The risk depends on the type of invalid arguments. For instance, using the `strcpy` function with a source argument larger than the destination argument can result in buffer overflows.

#### Fix

The fix depends on the standard library function involved in the defect. In some cases, you can constrain the function arguments before the function call. For instance, if the `strcpy` function:

```
char * strcpy(char * destination, const char* source)
```

tries to copy too many bytes into the destination argument compared to the available buffer, constrain the source argument before the call to `strcpy`. In some cases, you can use an alternative function to avoid the error. For instance, instead of `strcpy`, you can use `strncpy` to control the number of bytes copied.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Example — Invalid Use of Standard Library String Routine Error

```
 #include <string.h>
 #include <stdio.h>
 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

#### Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <string.h>
 #include <stdio.h>

 char* Copy_String(void)
 {
  char *res;
  /*Fix: gbuffer has equal or larger size than text */
  char gbuffer[20],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);

  return(res);
 }
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-806

# CWE Rule 824

Access of Uninitialized Pointer

## Description

### Rule Description

*The program accesses or uses a pointer that has not been initialized.*

### Polyspace Implementation

The rule checker checks for **Non-initialized pointer**.

## Examples

### Non-initialized pointer

#### Issue

This issue occurs when a pointer is not assigned an address before dereference.

#### Risk

Unless a pointer is explicitly assigned an address, it points to an unpredictable location.

#### Fix

The fix depends on the root cause of the defect. For instance, you assigned an address to the pointer but the assignment is unreachable.

Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

See examples of fixes below. It is a good practice to initialize a pointer to NULL when declaring the pointer.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications” if you review results in the Polyspace user interface.
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” if you review results in a web browser.
- “Annotate Code and Hide Known or Acceptable Results” if you review results in an IDE.

#### Extend Checker

If a pointer in your code is non-initialized only for certain system input values, you can see one possible combination of input values causing the defect. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

#### Example — Non-initialized pointer error

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j;
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

#### Correction — Initialize Pointer on Every Execution Path

One possible correction is to assign an address to `pi` when `prev` is not `NULL`.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
       {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
       }
    /* Fix: Initialize pi in branches of if statement  */
    else
        pi = prev;


    *pi = j;

    return pi;
}
```

## Check Information

**Category:** Pointer Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-824

# CWE Rule 825

Expired Pointer Dereference

## Description

### Rule Description

*The program dereferences a pointer that contains a location for memory that was previously valid, but is no longer valid.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Accessing object with temporary lifetime**
- **Deallocation of previously deallocated pointer**
- **Environment pointer invalidated by previous operation**
- **Pointer or reference to stack variable leaving scope**
- **Use of automatic variable as putenv-family function argument**
- **Use of previously freed pointer**

## Examples

### Accessing object with temporary lifetime

#### Issue

This issue occurs when you attempt to read from or write to an object with temporary lifetime that is returned by a function call. In a structure or union returned by a function, and containing an array, the array members are temporary objects. The lifetime of temporary objects ends:

- When the full expression or full declarator containing the call ends, as defined in the C11 Standard.
- After the next sequence point, as defined in the C90 and C99 Standards. A sequence point is a point in the execution of a program where all previous evaluations are complete and no subsequent evaluation has started yet.

For C++ code, **Accessing object with temporary lifetime** raises a defect only when you write to an object with a temporary lifetime.

If the temporary lifetime object is returned by address, no defect is raised.

#### Risk

Modifying objects with temporary lifetime is undefined behavior and can cause abnormal program termination and portability issues.

#### Fix

Assign the object returned from the function call to a local variable. The content of the temporary lifetime object is copied to the variable. You can now modify it safely.

### Example — Modifying Temporary Lifetime Object Returned by Function Call

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

/* func_temp() returns a struct value containing
* an array with a temporary lifetime.
*/
int func(void) {

/*Writing to temporary lifetime object is
 undefined behavior
 */
    return ++(func_temp().a[0]);
}

void main(void) {
    (void)func();
}
```

In this example, `func_temp()` returns by value a structure with an array member `a`. This member has temporary lifetime. Incrementing it is undefined behavior.

### Correction — Assign Returned Value to Local Variable Before Writing

One possible correction is to assign the return of the call to `func_temp()` to a local variable. The content of the temporary object `a` is copied to the variable, which you can safely increment.

```
 #include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define SIZE6 6

struct S_Array
{
    int t;
    int a[SIZE6];
};

struct S_Array func_temp(void);

int func(void) {

/* Assign object returned by function call to
```

```
 *local variable
 */
    struct S_Array s = func_temp();

/* Local variable can safely be
 *incremented
 */
    ++(s.a[0]);
    return s.a[0];
}

void main(void) {
    (void)func();
}
```

### Deallocation of previously deallocated pointer

**Issue**

This issue occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

**Risk**

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to free this block of memory can result in a segmentation fault.

**Fix**

The fix depends on the root cause of the defect. See if you intended to allocate a memory block to the pointer between the first deallocation and the second. Otherwise, remove the second `free` statement.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before freeing pointers, check them for NULL values and handle the error. In this way, you are protected against freeing an already freed block.

**Example — Deallocation of Previously Deallocated Pointer Error**

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

#### Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
 }
```

#### Example — Freeing Pointer Previously Reallocated With Possibly Zero Size

```
#include <stdlib.h>

void reshape(char *buf, size_t size) {
  char *reshaped_buf = (char *)realloc(buf, size);
  if (reshaped_buf == NULL) {
    free(buf);
  }
}
```

In this example, the argument `size` of the `reshape()` function can be zero and result in a zero-size reallocation with `realloc()`. In some implementations such as the GNU library, zero-size reallocations free the memory leading to a double free defect.

#### Correction – Guard Against Zero-Size Reallocations

One possible correction is to check size argument of `realloc()` for zero values before use. If the size argument is zero, you can simply free the memory instead of reallocating it.

```
#include <stdlib.h>

void reshape(char *buf, size_t size) {
  if (size != 0) {
    char *reshaped_buf = (char *)realloc(buf, size);
    if (reshaped_buf == NULL) {
      free(buf);
    }
  }
  else {
    free(buf);
  }

}
```

### Environment pointer invalidated by previous operation

#### Issue

This issue occurs when you use the third argument of *main()* in a hosted environment to access the environment after an operation modifies the environment. In a hosted environment, many C implementations support the nonstandard syntax:

```
main (int argc, char *argv[], char *envp[])
```

A call to a `setenv` or `putenv` family function modifies the environment pointed to by `*envp`.

**Risk**

When you modify the environment through a call to a `setenv` or `putenv` family function, the environment memory can potentially be reallocated. The hosted environment pointer is not updated and might point to an incorrect location. A call to this pointer can return unexpected results or cause an abnormal program termination.

**Fix**

Do not use the hosted environment pointer. Instead, use global external variable `environ` in Linux, `_environ` or `_wenviron` in Windows, or their equivalent. When you modify the environment, these variables are updated.

**Example — Access Environment Through Pointer envp**

```
#include <stdio.h>
#include <stdlib.h>

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

/* envp is from main function */
int func(char **envp)
{
    /* Call to setenv may cause environment
     *memory to be reallocated
     */
    if (setenv(("MY_NEW_VAR"),("new_value"),1) != 0)
    {
        /* Handle error */
        return -1;
    }
    /* envp not updated after call to setenv, and may
     *point to incorrect location.
     **/
    if (envp != ((void *)0)) {
        use_envp(envp);
/* No defect on second access to
*envp because defect already raised */
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func(envp);
    }
}
```

In this example, `envp` is accessed inside `func()` after a call to `setenv` that can reallocate the environment memory. `envp` can point to an incorrect location because it is not updated after `setenv` modifies the environment. No defect is raised when `use_envp()` is called because the defect is already raised on the previous line of code.

#### Correction — Use Global External Variable `environ`

One possible correction is to access the environment by using a variable that is always updated after a call to `setenv`. For instance, in the following code, the pointer `envp` is still available from `main()`, but the environment is accessed in `func()` through the global external variable `environ`.

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

extern int check_arguments(int argc, char **argv, char **envp);
extern void use_envp(char **envp);

int func(void)
{
    if (setenv(("MY_NEW_VAR"), ("new_value"),1) != 0) {
        /* Handle error */
        return -1;
    }
  /* Use global external variable environ
   *which is always updated after a call to setenv */

    if (environ != NULL) {
        use_envp(environ);
    }
    return 0;
}

void  main(int argc, char **argv, char **envp)
{
    if (check_arguments(argc, argv, envp))
    {
        (void)func();
    }
}
```

### Pointer or reference to stack variable leaving scope

#### Issue

This issue occurs when a pointer or reference to a local variable leaves the scope of the variable. For instance:

- A function returns a pointer to a local variable.
- A function performs the assignment `globPtr = &locVar`. `globPtr` is a global pointer variable and `locVar` is a local variable.
- A function performs the assignment `*paramPtr = &locVar`. `paramPtr` is a function parameter that is, for instance, an `int**` pointer and `locVar` is a local `int` variable.
- A C++ method performs the assignment `memPtr = &locVar`. `memPtr` is a pointer data member of the class the method belongs to. `locVar` is a variable local to the method.
- (C++11 and later) A function returns a lambda expression object that captures local variables of the function by reference.

The defect also applies to memory allocated using the `alloca` function. The defect does not apply to static, local variables. Polyspace assumes that the local objects within a function definition are in the same scope.

**Risk**

Local variables are allocated an address on the stack. Once the scope of a local variable ends, this address is available for reuse. Using this address to access the local variable value outside the variable scope can cause unexpected behavior.

If a pointer to a local variable leaves the scope of the variable, Polyspace Bug Finder highlights the defect. The defect appears even if you do not use the address stored in the pointer. For maintainable code, it is a good practice to not allow the pointer to leave the variable scope. Even if you do not use the address in the pointer now, someone else using your function can use the address, causing undefined behavior.

**Fix**

Do not allow a pointer or reference to a local variable to leave the variable scope.

**Example — Pointer to Local Variable Returned from Function**

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In this example, `func1` returns a pointer to local variable `ret`.

In `main`, `ptr` points to the address of the local variable. When `ptr` is accessed in `func2`, the access is illegal because the scope of `ret` is limited to `func1`,

**Example — Pointer to Local Variable Escapes Through Lambda Expression**

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [&] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

In this example, the `createAdder` function defines a lambda expression `adder` that captures the local variable `addThis` by reference. The scope of `addThis` is limited to the `createAdder` function. When the object returned by `createAdder` is called, a reference to the variable `addThis` is accessed outside its scope. When accessed in this way, the value of `addThis` is undefined.

#### Correction – Capture Local Variables by Copy in Lambda Expression Instead of Reference

If a function returns a lambda expression object, avoid capturing local variables by reference in the lambda object. Capture the variables by copy instead.

Variables captured by copy have the same lifetime as the lambda object, but variables captured by reference often have a smaller lifetime than the lambda object itself. When the lambda object is used, these variables accessed outside scope have undefined values.

```
auto createAdder(int amountToAdd) {
  int addThis = amountToAdd;
  auto adder = [=] (int initialAmount) {
      return (initialAmount + addThis);
  };
  return adder;
}

void func() {
  auto AddByTwo = createAdder(2);
  int res = AddByTwo(10);
}
```

### Use of automatic variable as putenv-family function argument

#### Issue

This issue occurs when the argument of a `putenv`-family function is a local variable with automatic duration.

#### Risk

The function `putenv(char *string)` inserts a pointer to its supplied argument into the environment array, instead of making a copy of the argument. If the argument is an automatic variable, its memory can be overwritten after the function containing the `putenv()` call returns. A subsequent call to `getenv()` from another function returns the address of an out-of-scope variable that cannot be dereferenced legally. This out-of-scope variable can cause environment variables to take on unexpected values, cause the program to stop responding, or allow arbitrary code execution vulnerabilities.

#### Fix

Use `setenv()`/`unsetenv()` to set and unset environment variables. Alternatively, use `putenv`-family function arguments with dynamically allocated memory, or, if your application has no reentrancy requirements, arguments with static duration. For example, a single thread execution with no recursion or interrupts does not require reentrancy. It cannot be called (reentered) during its execution.

#### Example — Automatic Variable as Argument of putenv()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    char env[SIZE1024];
    int retval = sprintf(env, "TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
        /* Handle error */
    }
    /* Environment variable TEST is set using putenv().
    The argument passed to putenv is an automatic variable. */
    retval = putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
}
```

In this example, `sprintf()` stores the character string `TEST=var` in `env`. The value of the environment variable `TEST` is then set to `var` by using `putenv()`. Because `env` is an automatic variable, the value of `TEST` can change once `func()` returns.

**Correction — Use `static` Variable for Argument of `putenv()`**

Declare `env` as a static-duration variable. The memory location of `env` is not overwritten for the duration of the program, even after `func()` returns.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024
void func(int var)
{
    /* static duration variable */
    static char env[SIZE1024];
    int retval = sprintf(env,"TEST=%s", var ? "1" : "0");
    if (retval <= 0) {
        /* Handle error */
    }

    /* Environment variable TEST is set using putenv() */
    retval=putenv(env);
    if (retval != 0) {
        /* Handle error */
    }
}
```

**Correction — Use `setenv()` to Set Environment Variable Value**

To set the value of `TEST` to `var`, use `setenv()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE1024 1024

void func(int var)
{
    /* Environment variable TEST is set using setenv() */
    int retval = setenv("TEST", var ? "1" : "0", 1);
    if (retval != 0) {
        /* Handle error */
    }
}
```

### Use of previously freed pointer

#### Issue

This issue occurs when you access a block of memory after freeing the block using the `free` function.

#### Risk

When a pointer is allocated dynamic memory with `malloc`, `calloc` or `realloc`, it points to a memory location on the heap. When you use the `free` function on this pointer, the associated block of memory is freed for reallocation. Trying to access this block of memory can result in unpredictable behavior or even a segmentation fault.

#### Fix

The fix depends on the root cause of the defect. See if you intended to free the memory later or allocate another memory block to the pointer before access.

As a good practice, after you free a memory block, assign the corresponding pointer to NULL. Before dereferencing pointers, check them for NULL values and handle the error. In this way, you are protected against accessing a freed block.

#### Example — Use of Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   { 
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift;
    /* Defect: Reading a freed pointer */
 
    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

#### Correction — Free Pointer After Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
#include <stdlib.h>

int increment_content_of_address(int base_val, int shift)
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

## Check Information

**Category:** Pointer Issues

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-825

# CWE Rule 839

Numeric Range Comparison Without Minimum Check

## Description

### Rule Description

*The program checks a value to ensure that it is less than or equal to a maximum, but it does not also verify that the value is greater than or equal to the minimum.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Pointer dereference with tainted offset**
- **Tainted sign change conversion**

## Examples

### Pointer dereference with tainted offset

#### Issue

This issue occurs when a pointer dereference uses an offset variable from an unknown or unsecure source.

This check focuses on dynamically allocated buffers. For static buffer offsets, see `Array access with tainted index`.

#### Risk

The index might be outside the valid array range. If the tainted index is outside the array range, it can cause:

- Buffer underflow/underwrite, or writing to memory before the beginning of the buffer.
- Buffer overflow, or writing to memory after the end of a buffer.
- Over reading a buffer, or accessing memory after the end of the targeted buffer.
- Under-reading a buffer, or accessing memory before the beginning of the targeted buffer.

An attacker can use an invalid read or write to compromise your program.

#### Fix

Validate the index before you use the variable to access the pointer. Check to make sure that the variable is inside the valid range and does not overflow.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

**Example — Dereference Pointer Array**

```
#include <stdio.h>
#include <stdlib.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if(pint) {
        /* Filling array */
        read_pint(pint);
        c = pint[offset];//Noncompliant
        free(pint);
    }
    return c;
}
```

In this example, the function initializes an integer pointer `pint`. The pointer is dereferenced using the input index `offset`. The value of `offset` could be outside the pointer range, causing an out-of-range error.

**Correction — Check Index Before Dereference**

One possible correction is to validate the value of `offset`. Continue with the pointer dereferencing only if `offset` is inside the valid range.

```
#include <stdlib.h>
#include <stdio.h>
enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};
extern void read_pint(int*);

int taintedptroffset(void) {
    int offset;
    scanf("%d",&offset);
    int* pint = (int*)calloc(SIZE10, sizeof(int));
    int c = 0;
    if (pint) {
        /* Filling array */
        read_pint(pint);
        if (offset>0 && offset<SIZE10) {
            c = pint[offset];
        }
        free(pint);
    }
    return c;
}
```

### Tainted sign change conversion

#### Issue

This issue occurs when values from unsecure sources are converted, implicitly or explicitly, from signed to unsigned values.

For example, functions that use `size_t` as arguments implicitly convert the argument to an unsigned integer. Some functions that implicitly convert `size_t` are:

```
bcmp
memcpy
memmove
strncmp
strncpy
calloc
malloc
memalign
```

#### Risk

If you convert a small negative number to unsigned, the result is a large positive number. The large positive number can create security vulnerabilities. For example, if you use the unsigned value in:

- Memory size routines — causes allocating memory issues.
- String manipulation routines — causes buffer overflow.
- Loop boundaries — causes infinite loops.

#### Fix

To avoid converting unsigned negative values, check that the value being converted is within an acceptable range. For example, if the value represents a size, validate that the value is not negative and less than the maximum value size.

#### Extend Checker

By default, Polyspace assumes that data from external sources are tainted. See “Sources of Tainting in a Polyspace Analysis”. To consider any data that does not originate in the current scope of Polyspace analysis as tainted, use the command line option `-consider-analysis-perimeter-as-trust-boundary`.

#### Example — Set Memory Value with Size Argument

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void bug_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size<SIZE128) {
        memset(str, 'c', size); //Noncompliant
    }
}
```

In this example, a `char` buffer is created and filled using `memset`. The size argument to `memset` is an input argument to the function.

The call to `memset` implicitly converts `size` to unsigned integer. If `size` is a large negative number, the absolute value could be too large to represent as an integer, causing a buffer overflow.

#### Correction — Check Value of `size`

One possible correction is to check if `size` is inside the valid range. This correction checks if `size` is greater than zero and less than the buffer size before calling `memset`.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum {
    SIZE10  =  10,
    SIZE100 = 100,
    SIZE128 = 128
};

void corrected_taintedsignchange(void) {
    int size;
    scanf("%d",&size);
    char str[SIZE128] = "";
    if (size>0 && size<SIZE128) {
        memset(str, 'c', size);
    }
}
```

## Check Information

**Category:** Numeric Errors

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

“Check for and Review Coding Standard Violations”

### External Websites

CWE-839

# CWE Rule 843

Access of Resource Using Incompatible Type ('Type Confusion')

## Description

### Rule Description

*The program allocates or initializes a resource such as a pointer, object, or variable using one type, but it later accesses that resource using a type that is incompatible with the original type.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Cast to pointer pointing to object of different type**
- **Reading memory reallocated from object of another type without reinitializing first**

## Examples

### Cast to pointer pointing to object of different type

#### Issue

The issue occurs when you perform a cast between a pointer to an object type and a pointer to a different object type.

#### Risk

If a pointer to an object is cast into a pointer to a different object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.

Even if the conversion produces a pointer that is correctly aligned, the behavior can be undefined if the pointer is used to access an object.

Exception: You can convert a pointer to object type into a pointer to one of these types:

- `char`
- `signed char`
- `unsigned char`

#### Example - Cast to Pointer Pointing to Object of Wider Type

```
signed   char *p1;
unsigned int *p2;

void foo(void){
  p2 = ( unsigned int * ) p1;     /* Non-compliant */
}
```

In this example, `p1` can point to a `signed char` object. However, `p1` is cast to a pointer that points to an object of wider type, `unsigned int`.

#### Example - Cast to Pointer Pointing to Object of Narrower Type

```
extern unsigned int read_value ( void );
extern void display ( unsigned int n );

void foo ( void ){
  unsigned int u = read_value ( );
  unsigned short *hi_p = ( unsigned short * ) &u;    /* Non-compliant  */
  *hi_p = 0;
  display ( u );
}
```

In this example, u is an `unsigned int` variable. &u is cast to a pointer that points to an object of narrower type, `unsigned short`.

On a big-endian machine, the statement `*hi_p = 0` attempts to clear the high bits of the memory location that &u points to. But, from the result of `display(u)`, you might find that the high bits have not been cleared.

#### Example - Cast Adding a Type Qualifier

```
const short *p;
const volatile short *q;
void foo (void){
  q = ( const volatile short * ) p;  /* Compliant */
}
```

In this example, both p and q can point to `short` objects. The cast between them adds a `volatile` qualifier only and is therefore compliant.

### Reading memory reallocated from object of another type without reinitializing first

#### Issue

This issue occurs when you do the following in sequence:

1. Reallocate memory to an object with a type that is different from the original allocation.

   For instance, in this code snippet, a memory originally allocated to a pointer with type `struct A*` is reallocated to a pointer with type `struct B*`:

   ```
   struct A;
   struct B;

   struct A *Aptr = (struct A*) malloc(sizeof(struct A));
   struct B *Bptr = (struct B*) realloc(Aptr, sizeof(struct B));
   ```
2. Read from this reallocated memory without reinitializing the memory first.

   Read accesses on the pointer to the reallocated memory can happen through pointer dereference or array indexing. Passing the pointer to a function that takes a pointer to a `const`-qualified object as the corresponding parameter also counts as a read access.

#### Risk

Reading from reallocated memory that has not been reinitialized leads to undefined behavior.

#### Fix

Reinitialize memory after reallocation and before the first read access.

The checker considers any write access on the pointer to the reallocated memory as satisfying the reinitialization requirement (even if the object might only be partially reinitialized). Write accesses on the pointer to the reallocated memory can happen through pointer dereference or array indexing. Passing the pointer to a function that takes a pointer to a non-`const`-qualified object as the corresponding parameter also counts as a write access.

**Example – Noncompliant: Reading from Reallocated Memory Without Reinitializing First**

```
#include<stdlib.h>

struct group {
    char *groupFirst;
    int groupSize;
};

struct groupWithID {
    int groupID;
    char *groupFirst;
    int groupSize;
};

char* readName();
int readSize();

void createGroup(int nextAvailableID) {
    struct group *aGroup;
    struct groupWithID *aGroupWithID;

    aGroup = (struct group*) malloc(sizeof(struct group));

    if(!aGroup) {
        /*Handle error*/
    }

    aGroup->groupFirst = readName();
    aGroup->groupSize  = readSize();

    /* Reassign to group with ID */
    aGroupWithID = (struct groupWithID*) realloc(aGroup, sizeof(struct groupWithID));
    if(!aGroupWithID) {
        free(aGroup);
        /*Handle error*/
    }

    if(aGroupWithID -> groupSize > 0) { /* Noncompliant */
        /* ... */
    }

    /* ...*/
    free(aGroupWithID);
}
```

In this example, the memory allocated to a `group*` pointer using the `malloc` function is reallocated to a `groupWithID*` pointer using the `realloc` function. There is a read access on the reallocated memory before the memory is reinitialized.

#### Correction – Reinitialize Memory After Reallocation and Before First Read

Reinitialize the memory assigned to the `groupWithID*` pointer before the first read access. All bits of the memory can be reinitialized using the `memset` function.

```
#include<stdlib.h>
#include<string.h>

struct group {
    char *groupFirst;
    int groupSize;
};

struct groupWithID {
    int groupID;
    char *groupFirst;
    int groupSize;
};

char* readName();
int readSize();

void createGroup(int nextAvailableID) {
    struct group *aGroup;
    struct groupWithID *aGroupWithID;

    aGroup = (struct group*) malloc(sizeof(struct group));

    if(!aGroup) {
        /*Handle error*/
    }

    aGroup->groupFirst = readName();
    aGroup->groupSize  = readSize();

    /* Reassign to group with ID */
    aGroupWithID = (struct groupWithID*) realloc(aGroup, sizeof(struct groupWithID));
    if(!aGroupWithID) {
        free(aGroup);
        /*Handle error*/
    }

    memset(aGroupWithID, 0 , sizeof(struct groupWithID));
    /* Reinitialize group */
    if(aGroupWithID -> groupSize > 0) {
        /* ... */
    }

    /* ...*/
    free(aGroupWithID);
}
```

## Check Information

**Category:** Type Errors

## Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-843

# CWE Rule 910

Use of Expired File Descriptor

## Description

### Rule Description

*The software uses or accesses a file descriptor after it has been closed.*

### Polyspace Implementation

The rule checker checks for **Use of previously closed resource**.

## Examples

### Use of previously closed resource

#### Issue

This issue occurs when a function operates on a stream that you closed earlier in your code.

#### Risk

The standard states that the value of a `FILE*` pointer is indeterminate after you close the stream associated with it. Operations using the `FILE*` pointer can produce unintended results.

#### Fix

One possible fix is to close the stream only at the end of operations. Another fix is to reopen the stream before using it again.

#### Example — Use of FILE* Pointer After Closing Stream

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fclose(fp);
        fprintf(fp,"text");
    }
}
```

In this example, `fclose` closes the stream associated with `fp`. When you use `fprintf` on `fp` after `fclose`, the **Use of previously closed resource** defect appears.

#### Correction — Close Stream After All Operations

One possible correction is to reverse the order of the `fprintf` and `fclose` operations.

```
#include <stdio.h>
```

```
void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fprintf(fp,"text");
        fclose(fp);
    }
}
```

## Check Information

**Category:** Resource Management Errors

# Version History

**Introduced in R2023a**

## See Also

Check CWE (-cwe)

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-910

# CWE Rule 922

Insecure Storage of Sensitive Information

## Description

### Rule Description

*The software stores sensitive information without properly limiting read or write access by unauthorized actors.*

### Polyspace Implementation

The rule checker checks for these issues:

- **File manipulation after chroot() without chdir("/")**
- **Umask used with chmod-style arguments**
- **Use of non-secure temporary file**
- **Vulnerable permission assignments**

## Examples

### File manipulation after chroot() without chdir("/")

#### Issue

This issue occurs when you have access to a file system outside of the jail created by `chroot`. By calling `chroot`, you create a file system jail that confines access to a specific file subsystem. However, this jail is ineffective if you do not call `chdir("/")`.

#### Risk

If you do not call `chdir("/")` after creating a `chroot` jail, file manipulation functions that takes a path as an argument can access files outside of the jail. An attacker can still manipulate files outside the subsystem that you specified, making the chroot jail ineffective.

#### Fix

After calling `chroot`, call `chdir("/")` to make your `chroot` jail more secure.

#### Example — Open File in chroot-jail

```
#include <unistd.h>
#include <stdio.h>

const char root_path[] = "/var/ftproot";
const char log_path[] = "file.log";
FILE* chrootmisuse() {
    FILE* res;
    chroot(root_path);
    chdir("base");
    res = fopen(log_path, "r");
    return res;
}
```

This example uses `chroot` to create a chroot-jail. However, to use the `chroot` jail securely, you must call `chdir("\")` afterward. This example calls `chdir("base")`, which is not equivalent. Bug Finder also flags `fopen` because `fopen` opens a file in the vulnerable `chroot`-jail.

**Correction — Call `chdir("/")`**

Before opening files, call `chdir("/")`.

```
#include <unistd.h>
#include <stdio.h>

const char root_path[] = "/var/ftproot";
const char log_path[] = "file.log";
FILE* chrootmisuse() {
    FILE* res;
    chroot(root_path);
    chdir("/");
    res = fopen(log_path, "r");
    return res;
}
```

### Umask used with chmod-style arguments

**Issue**

This issue occurs when `umask` commands have arguments specified in the style of arguments to `chmod` and provide possibly unintended permissions. For instance:

- The `umask` command provides more permissions to the group than the current user.
- The `umask` command provides more permissions to other users than the group.

For new files, the `umask` argument or the mask value specifies which permissions *not* to set, in other words, which permissions to remove. The mask is bitwise-negated and then applied to new file permissions. In contrast, `chmod` sets the permissions as you specify them.

**Risk**

If you use `chmod`-style arguments, you specify opposite permissions of what you want. This mistake can give external users unintended read/write access to new files and folders.

**Fix**

To fix this defect, perform both of these tasks:

- Enable read permission for the user.
- Set the mask value so that the user (`u`) has equal or fewer permissions turned off than the group (`g`) and the group has equal or fewer permissions turned off than other users (`o`), or `u <= g <= o`.

You can see the umask value by calling,

```
umask
```

or the symbolic value by calling,

```
umask -S
```

**Example — Setting the Default Mask**

```
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>

typedef mode_t (*umask_func)(mode_t);

const mode_t default_mode = (
    S_IRUSR    /* 00400 */
    | S_IWUSR  /* 00200 */
    | S_IRGRP  /* 00040 */
    | S_IWGRP  /* 00020 */
    | S_IROTH  /* 00004 */
    | S_IWOTH  /* 00002 */
    );         /* 00666 (i.e. -rw-rw-rw-) */

static void my_umask(mode_t mode)
{
    umask(mode);
}

int umask_use(mode_t m)
{
    my_umask(default_mode);
    return 0;
}
```

This example uses a function called `my_umask` to set the default mask mode. However, the `default_mode` variable gives the permissions 666 or `-rw-rw-rw`. `umask` negates this value. However, this negation means the default mask mode turns off read/write permissions for the user, group users, and other outside users.

**Correction — Negate Preferred Permissions**

One possible correction is to negate the `default_mode` argument to `my_umask`. This correction nullifies the negation `umask` for new files.

```
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>

typedef mode_t (*umask_func)(mode_t);

const mode_t default_mode = (
    S_IRUSR    /* 00400 */
    | S_IWUSR  /* 00200 */
    | S_IRGRP  /* 00040 */
    | S_IWGRP  /* 00020 */
    | S_IROTH  /* 00004 */
    | S_IWOTH  /* 00002 */
    );         /* 00666 (i.e. -rw-rw-rw-) */

static void my_umask(mode_t mode)
{
    umask(mode);
}

int umask_use(mode_t m)
{
    my_umask(~default_mode);
    return 0;
}
```

### Use of non-secure temporary file

#### Issue

This issue occurs when you use temporary file routines that are not secure.

#### Risk

If an attacker guesses the file name generated by a standard temporary file routine, the attacker can:

- Cause a race condition when you generate the file name.
- Precreate a file of the same name, filled with malicious content. If your program reads the file, the attacker's file can inject the malicious code.
- Create a symbolic link to a file storing sensitive data. When your program writes to the temporary file, the sensitive data is deleted.

#### Fix

To create temporary files, use a more secure standard temporary file routine, such as `mkstemp` from POSIX.1-2001.

Also, when creating temporary files with routines that allow flags, such as `mkostemp`, use the exclusion flag `O_EXCL` to avoid race conditions.

#### Example — Temp File Created With tempnam

```
#define _BSD_SOURCE
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int test_temp()
{
    char tpl[] = "abcXXXXXX";
    char suff_tpl[] = "abcXXXXXXsuff";
    char *filename = NULL;
    int fd;

    filename = tempnam("/var/tmp", "foo_");

    if (filename != NULL)
    {
        printf("generated tmp name (%s) in (%s:%s:%s)\n",
               filename, getenv("TMPDIR") ? getenv("TMPDIR") : "$TMPDIR",
                 "/var/tmp", P_tmpdir);

        fd = open(filename, O_CREAT, S_IRWXU|S_IRUSR);
        if (fd != -1)
        {
            close(fd);
            unlink(filename);
            return 1;
        }
    }
    return 0;
}
```

In this example, Bug Finder flags `open` because it tries to use an unsecure temporary file. The file is opened without exclusive privileges. An attacker can access the file causing various risks on page 13-84.

#### Correction — Add O_EXCL Flag

One possible correction is to add the `O_EXCL` flag when you open the temporary file.

```
#define _BSD_SOURCE
#define _XOPEN_SOURCE
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int test_temp()
{
    char tpl[] = "abcXXXXXX";
    char suff_tpl[] = "abcXXXXXXsuff";
    char *filename = NULL;
    int fd;

    filename = tempnam("/var/tmp", "foo_");

    if (filename != NULL)
    {
        printf("generated tmp name (%s) in (%s:%s:%s)\n",
                 filename, getenv("TMPDIR") ? getenv("TMPDIR") : "$TMPDIR",
                 "/var/tmp", P_tmpdir);

        fd = open(filename, O_CREAT|O_EXCL, S_IRWXU|S_IRUSR);
        if (fd != -1)
        {
            close(fd);
            unlink(filename);
            return 1;
        }
    }
    return 0;
}
```

### Vulnerable permission assignments

**Issue**

This issue occurs when functions that can change resource permissions, such as `chmod`, `umask`, `creat`, or `open`, specify permissions that allow unintended actors to modify or read the resource.

**Risk**

If you give outside users or outside groups a wider range or permissions than required, you potentially expose your sensitive information and your modifications. This defect is especially dangerous for permissions related to:

- Program configurations
- Program executions
- Sensitive user data

**Fix**

Set your permissions so that the user (`u`) has more permissions than the group (`g`), and so the group has more permissions than other users (`o`), or `u >= g >= o`.

**Example — Create File with Other Permissions**

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void bug_dangerouspermissions(const char * log_path) {
    mode_t mode = S_IROTH | S_IXOTH | S_IWOTH;
    int fd = creat(log_path, mode);

    if (fd) {
        write(fd, "Hello\n", 6);
    }
    close(fd);
    unlink(log_path);
}
```

In this example, the `log_path` file is created with more rights for the other outside users, than the current user. The permissions are `---------rwx`.

**Correction — Modify User Permissions**

One possible correction is to modify the user permissions for the file. In this correction, the user has read/write/execute permissions, but other users do not.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void corrected_dangerouspermissions(const char * log_path) {
    mode_t mode = S_IRUSR | S_IXUSR | S_IWUSR;
    int fd = creat(log_path, mode);

    if (fd) {
        write(fd, "Hello\n", 6);
    }
    close(fd);
    unlink(log_path);
}
```

### Check Information

**Category:** Others

## Version History

**Introduced in R2023a**

### See Also

`Check CWE (-cwe)`

#### Topics

"Check for and Review Coding Standard Violations"

#### External Websites

CWE-922

# CWE Rule 1071

Empty code block

## Description

### Rule Definition

*The source code contains a block that does not contain any code, i.e., the block is empty.*

### Polyspace Implementation

This rule checker checks for the issue **Empty Code Block**.

## Examples

### Empty Code Block

#### Issue

**Empty Code Block** occurs when a code block does not contain code.

#### Risk

Unintentional empty code block can occur if you do not complete implementation or if you accidentally delete code. Unintentional missing code can cause errors or unexpected results.

#### Fix

Remove the empty code block if possible. Fill in the missing or deleted code if the code block is unintentionally empty due to accidental code deletion or incomplete implementation. If you can't remove or implement missing code, an empty code block can be replaced by an empty instruction. For example:

```
while (true)
;
```

#### Example — Empty Code Block

In this example, the `for` loop code block is empty.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> a = { 0, 1, 2, 4, 8 };

    for (auto x : a)                //Noncompliant
    {

    }

    return 0;
}
```

#### Correction — Add Missing Code

By adding code to the `for` loop code block, the example becomes compliant. Alternatively, if you do not need the `for` loop, remove it from the code.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> a = { 0, 1, 2, 4, 8 };

    for (auto x : a)                            //Compliant
    {
        std::cout << x << std::endl;
    }

    return 0;
}
```

## Check Information

**Category:** Bad Coding Practices

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-1071

# CWE Rule 1335

Incorrect Bitwise Shift of Integer

## Description

### Rule Description

*An integer value is specified to be shifted by a negative amount or an amount greater than or equal to the number of bits contained in the value causing an unexpected or indeterminate result.*

### Polyspace Implementation

The rule checker checks for these issues:

- **Right operand of shift operation outside allowed bounds**
- **Shift of a negative value**

## Examples

### Right operand of shift operation outside allowed bounds

#### Issue

This issue occurs when a shift operation can result in values that cannot be represented by the result data type. The data type of a variable determines the number of bytes allocated for the variable storage and constrains the range of allowed values.

The exact storage allocation for different data types depends on your processor. See `Target processor type (-target)`.

#### Risk

Shift operation overflows can result in undefined behavior.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also “Interpret Bug Finder Results in Polyspace Desktop User Interface” or “Interpret Bug Finder Results in Polyspace Access Web Interface”.

You can fix the defect by:

- Using a bigger data type for the result of the shift operation so that all values can be accommodated.
- Checking for values that lead to the overflow and performing appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

#### Example — Left Shift of Integer

```
int left_shift(void) {

    int foo = 33;
    return 1 << foo;
}
```

In the return statement of this function, bit-wise shift operation is performed shifting 1 `foo` bits to the left. However, an `int` has only 32 bits, so the range of the shift must be between 0 and 31. Therefore, this shift operation causes an overflow.

#### Correction — Different storage type

One possible correction is to store the shift operation result in a larger data type. In this example, by returning a `long long` instead of an `int`, the overflow defect is fixed.

```
long long left_shift(void) {

    int foo = 33;
    return 1LL << foo;
}
```

### Shift of a negative value

#### Issue

This issue occurs when a bit-wise shift is used on a variable that can have negative values.

#### Risk

Shifts on negative values overwrite the sign bit that identifies a number as negative. The shift operation can result in unexpected values.

#### Fix

The fix depends on the root cause of the defect. Often the result details (or source code tooltips in Polyspace as You Code) show a sequence of events that led to the defect. You can implement the fix on any event in the sequence. If the result details do not show this event history, you can search for previous references of variables relevant to the defect using right-click options in the source code and find related events. See also "Interpret Bug Finder Results in Polyspace Desktop User Interface" or "Interpret Bug Finder Results in Polyspace Access Web Interface".

To fix the defect, check for negative values before the bit-wise shift operation and perform appropriate error handling.

See examples of fixes below.

If you do not want to fix the issue, add comments to your result or code to avoid another review. See:

- "Address Results in Polyspace User Interface Through Bug Fixes or Justifications" if you review results in the Polyspace user interface.
- "Address Results in Polyspace Access Through Bug Fixes or Justifications" if you review results in a web browser.
- "Annotate Code and Hide Known or Acceptable Results" if you review results in an IDE.

#### Extend Checker

A default Bug Finder analysis might not raise this defect when the input values are unknown and only a subset of inputs cause an issue. To check for defects caused by specific system input values, run a stricter Bug Finder analysis. See "Extend Bug Finder Checkers to Find Defects from Specific System Input Values".

#### Example — Shifting a negative variable

```
int shifting(int val)
{
    int res = -1;
    return res << val;
}
```

In the return statement, the variable `res` is shifted a certain number of bits to the left. However, because `res` is negative, the shift might overwrite the sign bit.

#### Correction — Change the Data Type

One possible correction is to change the data type of the shifted variable to unsigned. This correction eliminates the sign bit, so left shifting does not change the sign of the variable.

```
int shifting(int val)
{
    unsigned int res = -1;
    return res << val;
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

#### Topics

"Check for and Review Coding Standard Violations"

**External Websites**
CWE-1335

# CWE Rule 1341

Multiple Releases of Same Resource or Handle

## Description

### Rule Description

*The product attempts to close or release a resource or handle more than once, without any successful open between the close operations.*

### Polyspace Implementation

The rule checker checks for **Closing previously closed resource**.

## Examples

### Closing previously closed resource

#### Issue

This issue occurs when a function attempts to close a stream that was closed earlier in your code and not reopened later.

#### Risk

The standard states that the value of a `FILE*` pointer is indeterminate after you close the stream associated with it. Performing the close operation on the `FILE*` pointer again can cause unwanted behavior.

#### Fix

Remove the redundant close operation.

#### Example — Closing Previously Closed Resource

```
#include <stdio.h>

void func(char* data) {
    FILE* fp = fopen("file.txt", "w");
    if(fp!=NULL) {
        if(data)
            fputc(*data,fp);
        else
            fclose(fp);
    }
    fclose(fp);
}
```

In this example, if `fp` is not `NULL` and `data` is `NULL`, the `fclose` operation occurs on `fp` twice in succession.

**Correction — Remove Close Operation**

One possible correction is to remove the last `fclose` operation. To avoid a resource leak, you must also place an `fclose` operation in the `if(data)` block.

```
#include <stdio.h>

void func(char* data) {
    FILE* fp = fopen("file.txt", "w");
    if(fp!=NULL) {
        if(data) {
            fputc(*data,fp);
            fclose(fp);
        }
        else
            fclose(fp);
    }
}
```

## Check Information

**Category:** Others

# Version History

**Introduced in R2023a**

## See Also

`Check CWE (-cwe)`

### Topics

"Check for and Review Coding Standard Violations"

### External Websites

CWE-1341

28

# Guidelines

## Software Complexity

# Number of calling functions exceeds threshold

The number of distinct callers of a function is greater than the defined threshold

## Description

This defect is raised on a function when the number of distinct callers of the function is greater than the defined checker threshold. For details about how Polyspace calculates the number of calling functions, see `Number of Calling Functions`.

Polyspace uses the default threshold 5 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see “Reduce Software Complexity by Using Polyspace Checkers”.

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Calling Functions` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that:

- Issues arising in the flagged function might propagate to have unexpected impact in many places.
- The data flow in the file is convoluted and finding the root cause of an issue might be difficult.
- The flagged function might be performing multiple tasks.

These factors make the module difficult to maintain and debug.

### Fix

To fix this check, either refactor your code or change the checker threshold. When refactoring the code, design the functions in your code so that:

- Each function performs one specific task.
- The functions have minimal side effects on other functions.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Functions That are Called by Many Different Functions

```
void utilityFunc(){//Noncompliant
    //...
}

void task1(){
    utilityFunc();
    //...
}
void task2(){
    utilityFunc();
    //...
}
void task3(){
    utilityFunc();
    //...
}
void task4(){
    utilityFunc();
    //...
}
void task5(){
    utilityFunc();
    //...
}
void task6(){
    utilityFunc();
    //...
}
void task7(){
    utilityFunc();
    //...
}
```

In this example, the function `utilityFunc` is called by seven different functions. It might indicate that `utilityFunc` performs multiple tasks. An issue arising in `utilityFunc` might have unexpected impact in any of the seven function that calls `utilityFunc`. Such interdependency make the code difficult to maintain and debug. Polyspace flags the function `utilityFunc` as noncompliant.

#### Correction — Refactor the Function

One possible correction is to refactor the function. For instance, delegate the different tasks required by different tasks into two utility functions instead of one.

```
void utilityFuncA(){//Compliant
    //...
}
void utilityFuncB(){//Compliant
    //...
}

void task1(){
    utilityFuncA();
    //...
}
void task2(){
    utilityFuncA();
    //...
}
void task3(){
    utilityFuncA();
    //...
}
void task4(){
    utilityFuncB();
    //...
}
void task5(){
    utilityFuncB();
    //...
}
void task6(){
    utilityFuncB();
    //...
}
void task7(){
    utilityFuncB();
    //...
}
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** `SC04`
**Default Threshold:** 5

# Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Number of called functions exceeds threshold

The number of distinct function calls within the body of a function is greater than the defined threshold

## Description

This defect is raised on a function when the number of calls to distinct functions within its body is greater than the defined checker threshold. For details about how Polyspace calculates the number of called functions, see `Number of Called Functions`.

Polyspace uses the default threshold 7 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see "Reduce Software Complexity by Using Polyspace Checkers".

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Called Functions` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that:

- The function depends on many functions and changes to any one of these functions might result in an unexpected behavior.
- The function performs too many tasks.
- The module might contain unexpected or unplanned development.

These factors make the module difficult to maintain and debug.

### Fix

To fix this check, either refactor your code or change the checker threshold. When refactoring the code, design the functions in your code so that:

- Each function performs one specific task.
- The functions have minimal side effects on other functions.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Functions That Call Too Many Functions

```
class A{
    //..
};
class B{
    //...
};
void transform1(A &a, B &b){
    //...
}
void transform2(A &a, B &b){
    //...
}
void transform3(A &a, B &b){
    //...
}
void transform4(A &a, B &b){
    //...
}
void transform5(A &a, B &b){
    //...
}
void transform6(A &a, B &b){
    //...
}
void transform7(A &a, B &b){
    //...
}
void transform8(A &a, B &b){
    //...
}
void main(){//Noncompliant
    A a;
    B b;
    transform1(a,b);
    transform2(a,b);
    transform3(a,b);
    transform4(a,b);
    transform5(a,b);
    transform6(a,b);
    transform7(a,b);
    transform8(a,b);
}
```

In this example, the function `main()` calls eight different functions which is greater than the defined threshold of function calls. Debugging `main()` is difficult because the issue might be caused by any of these eight functions. Polyspace flags the function `main()`.

#### Correction — Design Functions to Perform Specific Tasks

One possible correction is design functions that perform specific tasks. In this case, the individual tasks are already delegated to different functions, but `main()` performs too may of these tasks. Dividing the task load of main between multiple functions might make the functions easier to debug, maintain, and test.

```
class A{
    //..
};
class B{
    //...
};
void transform1(A &a, B &b){
    //...
}
void transform2(A &a, B &b){
    //...
}
void transform3(A &a, B &b){
    //...
}
void transform4(A &a, B &b){
    //...
}
void transform5(A &a, B &b){
    //...
}
void transform6(A &a, B &b){
    //...
}
void transform7(A &a, B &b){
    //...
}
void transform8(A &a, B &b){
    //...
}
void FirstStep(A &a, B &b){
    transform1(a,b);
    transform2(a,b);
    transform3(a,b);
}
void SecondStep(A &a, B &b){
    transform4(a,b);
    transform5(a,b);
    transform6(a,b);
}
void ThirdStep(A &a, B &b){
    transform7(a,b);
    transform8(a,b);
}
void main(){//Compliant
    A a;
    B b;
    FirstStep(a,b);
    SecondStep(a,b);
    ThirdStep(a,b);
}
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC05
**Default Threshold:** 7

## Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

“Reduce Software Complexity by Using Polyspace Checkers”
“Modify Default Behavior of Bug Finder Checkers”

# Comment density falls below threshold

The comment density of the module falls below the specified threshold

## Description

Polyspace calculates the comment density percentage of a file by taking the ratio of comments to number of executable lines in the file and then multiplying the ratio by 100. For instance, a comment density `20` indicates that the file contains 20% comments and 80% code statements. This defect is raised when the comment density falls below the specified threshold. For details about how Polyspace calculates comment density, see `Comment Density`

Polyspace uses the default threshold 80 unless you specify a threshold. Use `Set checkers by file (-checkers-selection-file)` to specify a selection file where you can set the threshold. See “Reduce Software Complexity by Using Polyspace Checkers”.

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Comment Density` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that:

- The module is not properly documented.
- The module is overly long and dense.
- The module might contain unexpected or unplanned development.

These factors make the module difficult to maintain and debug.

### Fix

To fix this defect:

- Put more comments between your statements that conveys the developer intent.
- Alternatively, reduce the number of statements in the module. If the module contains multiple functions, consider splitting the module and documenting them separately.

To fix this check, either refactor your code or change the checker threshold. A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Poorly Documented Code

```
//File1.cpp        //Noncompliant
 long long power(double x, int n){
     long long BN = 1;
     for(int i = 0; i<n;++i){
         BN*=x;
     }
     return BN;
 }

 double AppxIndex(double m, double f){
     double U = (power(m,2) - 1)/(power(m,2)+2);
     double V = (power(m,4) + 27*power(m,2)+38)/(2*power(m,2)+3);
     return (1+2*f*power(U,2)*(1+power(m,2)*U*V +
                power(m,3)/power(m,3)*(U-V)))/( (1-2*f*power(U,2)
              *(1+power(m,2)*U*V + power(m,3)/power(m,3)*(U-V))));
 }
```

In this example, the code in `File1.cpp` does not contain sufficient comments to document the developer intent. lack of sufficient documentation makes the code difficult to understand. Subsequent debugging and maintenance of the code might be difficult. Polyspace flags the file as noncompliant to this rule.

#### Correction — Document the Developer intent with Comments

One possible correction is to document the developer intent by putting sufficient comments. This code contains comment that clearly document developers intent and conveys information that eases understanding, debugging, and maintaining the code.

```
//File1.c        //Compliant
// r = power(x,n) returns the nth power of x into y
// r is long long
// x is double
// n must be integer
 long long power(double x, int n){
     long long BN = 1;
     for(int i = 0; i<n;++i){
         BN*=x;
     }
     return BN;
 }
 // n = AppxIndex(m,f) calculates the approximate
//  effective index of a material
// n is a double, represent the effective index of a mixture
// m is a double, represents the relative index of the
//                  inclusion compared to the background material
// f is a double, represents the filling factor of the inclusion
// The function implements the formula in the doi 10.XXYY
 double AppxIndex(double m, double f){
     // This function implements the formula for approximate index
     // The first term is U:
     double U = (power(m,2) - 1)/(power(m,2)+2);
     //The second term is V:
     double V = (power(m,4) + 27*power(m,2)+38)/(2*power(m,2)+3);
     // Calculate the denominator
     double den = ( (1-2*f*power(U,2)*
              (1+power(m,2)*U*V + power(m,3)/power(m,3)*(U-V))));
     // Calculate the numerator
     double num = (1+2*f*power(U,2)
              *(1+power(m,2)*U*V + power(m,3)/power(m,3)*(U-V)));
     // Calculate the aproximate index
     // Return by value
```

```
    return num/den;
}
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** `SC02`
**Default Threshold:** 20

# Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

“Reduce Software Complexity by Using Polyspace Checkers”
“Modify Default Behavior of Bug Finder Checkers”

# Call tree complexity exceeds threshold

The call tree complexity of a file is greater than the defined threshold

## Description

The call tree complexity of a file represents the complexity between different levels of its function call tree. Polyspace calculates the call tree complexity of a file as:

$$\textit{Call tree Complexity} = \textit{number of call occurrences} - \textit{number of function definitions} + 1 \qquad (28\text{-}1)$$

This defect is raised when the calculated call tree complexity of a file exceeds the defined call tree complexity threshold of the file. For details about how Polyspace calculates call tree complexity, see `Estimated Function Coupling`

Polyspace uses the default threshold 20 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see “Reduce Software Complexity by Using Polyspace Checkers”.

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Estimated Function Coupling` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that:

- The interdependency of the functions in the file is unacceptably high.
- Changes in one function in the file might introduce bugs or unexpected behavior in other functions in the file.
- Reusing a single function of the file might be difficult because of interdependencies with other functions.

These factors make the file difficult to maintain, test, and debug.

### Fix

To fix this check, either refactor your code or change the checker threshold. When refactoring the code, design the functions in your code so that:

- Each function performs one specific task.
- The functions have minimal side effects on other functions.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Overly Complex Function Call Tree

```
//Noncompliant
 long long power(double x, int n){
     long long BN = 1;
     for(int i = 0; i<n;++i){
         BN*=x;
     }
     return BN;
 }

 double AppxIndex(double m, double f){
     double U = (power(m,2) - 1)/(power(m,2)+2);
     double V = (power(m,4) + 27*power(m,2)+38)/(2*power(m,2)+3);
     return (1+2*f*power(U,2)*(1+power(m,2)*U*V + power(m,3)/
         power(m,3)*(U-V)))/( (1-2*f*power(U,2)*(1+power(m,2)*U*V
        + power(m,3)/power(m,3)*(U-V))));
 }
```

In this example, the function `power` is called many times in the function `AppIndex`. The high interdependency between these two functions results in a call tree complexity of 12, which is greater than the threshold 10. This high value implies that changes in `power` might require rewriting or revising `AppIndex`. Polyspace flags the file as noncompliant.

#### Correction — Design Functions to Perform One Specific Task

One possible correction is to refactor the code so that a function performs one specific task. In this code, the functions have a lower interdependency. For instance, `CalculateU` and `CalculateV` are completely independent of each other. By designing the functions to perform one specific task, isolating any unexpected behavior is easier. This code is easier to debug, test, and maintain.

```
//Compliant
 long long power(double x, int n){
     long long BN = 1;
     for(int i = 0; i<n;++i){
         BN*=x;
     }
     return BN;
 }
 double CalculateU(double m){
     return (power(m,2) - 1)/(power(m,2)+2);
 }
 double CalculateV(double m){//Compliant
     return (power(m,4) + 27*power(m,2)+38)/(2*power(m,2)+3);
 }
 double CalculateMid(double m, double f){//Compliant
     double U = CalculateU(m);
     double V = CalculateU(m);
     return 2*f*power(U,2)*(1+power(m,2)*U*V
              + power(m,3)/power(m,3)*(U-V));
 }
 double AppxIndex(double m, double f){//Compliant
     return (1+CalculateMid(m,f))/( (1-CalculateMid(m,f)));
 }
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** `SC03`
**Default Threshold:** 20

## Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Number of lines in function body exceeds threshold

The number of lines in the body of a function is greater than the defined threshold

## Description

This defect is raised on a function when the number of lines within the body of the function is greater than the defined checker threshold. For details about how Polyspace calculates the number of lines in the body of a function, see `Number of Lines Within Body`

Polyspace uses the default threshold 1200 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see “Reduce Software Complexity by Using Polyspace Checkers”.

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Lines Within Body` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that the function is too long. Lengthy function are difficult to read and understand. Maintaining, testing and debugging lengthy functions might be costly in terms of resource and time.

### Fix

To fix this check, either refactor your code or change the checker threshold. When refactoring the code, make the functions modular. That is, design your code so that each function performs one specific task with as little side effect as possible. Modular design of function make them easy to test, debug, and maintain. Modular functions also enables efficient code reuse and might reduce code duplication.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Lengthy Functions

```
#include <cmath>
#include <math.h>
#include <stddef.h>
#include <stdlib.h>
#define CREAL_T
typedef float real32_T;
typedef double real64_T;
typedef struct {
  real32_T re;
  real32_T im;
} creal32_T;

typedef struct {
  real64_T re;
  real64_T im;
} creal_T;
// Function Declarations
static double rt_powd_snf(double u0, double u1);

void CalculateAppxIndex(double r, double epsilon_s, double epsilon_h, double//Noncompliant
  lambda, double f, creal_T *eps_eff, creal_T *mu_eff)
{
  double n_h;
  double m;
  double a;
  double alpha_e_re;
  double alpha_e_im;
  double alpha_m_re;
  double alpha_m_im;
  int l;
  double br;

  //  sphere radius is 1 micron;
  //  the refractive index of inclusion
  n_h = std::sqrt(epsilon_h);

  //  the refractive index of host
  m = std::sqrt(epsilon_s) / n_h;
  n_h = 6.2831853071795862 * n_h * r / lambda;

  //  size parameter;
  a = (n_h + m) + 2.0;
  m = n_h - m;

  //  polarizability
  n_h = 6.2831853071795862 * rt_powd_snf(r, 3.0) / rt_powd_snf(n_h, 3.0);
  alpha_e_re = 0.0;
  alpha_e_im = 0.0;
  alpha_m_re = 0.0;
  alpha_m_im = 0.0;
  for (l = 0; l < 5; l++) {
    alpha_e_re += 0.0 * ((2.0 * (1.0 + (double)l) + 1.0) * a);
    alpha_e_im += (2.0 * (1.0 + (double)l) + 1.0) * a;
    alpha_m_re += 0.0 * ((2.0 * (1.0 + (double)l) + 1.0) * (m + 2.0));
    alpha_m_im += (2.0 * (1.0 + (double)l) + 1.0) * (m + 2.0);

    //  alpha = alpha + 1i* [(2*l+1)*(an(l) + bn(l))];
  }

  alpha_e_re *= n_h;
  alpha_e_im *= n_h;
  alpha_m_re *= n_h;
  alpha_m_im *= n_h;

  // alpha = aa*alpha;
  n_h = f / (4.1887902047863905 * rt_powd_snf(r, 3.0));
```

```
alpha_e_re *= n_h;
alpha_e_im *= n_h;
alpha_m_re *= n_h;
alpha_m_im *= n_h;
if (alpha_e_im == 0.0) {
  m = alpha_e_re / 3.0;
  n_h = 0.0;
} else if (alpha_e_re == 0.0) {
  m = 0.0;
  n_h = alpha_e_im / 3.0;
} else {
  m = alpha_e_re / 3.0;
  n_h = alpha_e_im / 3.0;
}

br = 1.0 - m;
m = 0.0 - n_h;
if (m == 0.0) {
  if (alpha_e_im == 0.0) {
    m = alpha_e_re / br;
    alpha_e_im = 0.0;
  } else if (alpha_e_re == 0.0) {
    m = 0.0;
    alpha_e_im /= br;
  } else {
    m = alpha_e_re / br;
    alpha_e_im /= br;
  }
} else {
  n_h = std::abs(m);
  if (br > n_h) {
    a = m / br;
    n_h = br + a * m;
    m = (alpha_e_re + a * alpha_e_im) / n_h;
    alpha_e_im = (alpha_e_im - a * alpha_e_re) / n_h;
  } else if (n_h == br) {
    if (br > 0.0) {
      a = 0.5;
    } else {
      a = -0.5;
    }

    if (m > 0.0) {
      n_h = 0.5;
    } else {
      n_h = -0.5;
    }

    m = alpha_e_re * a + alpha_e_im * n_h;
    alpha_e_im = alpha_e_im * a - alpha_e_re * n_h;
  } else {
    a = br / m;
    n_h = m + a * br;
    m = (a * alpha_e_re + alpha_e_im) / n_h;
    alpha_e_im = (a * alpha_e_im - alpha_e_re) / n_h;
  }
}
```

```
eps_eff->re = epsilon_h * (1.0 + m);
eps_eff->im = epsilon_h * alpha_e_im;
if (alpha_m_im == 0.0) {
  m = alpha_m_re / 3.0;
  n_h = 0.0;
} else if (alpha_m_re == 0.0) {
  m = 0.0;
  n_h = alpha_m_im / 3.0;
} else {
  m = alpha_m_re / 3.0;
  n_h = alpha_m_im / 3.0;
}

br = 1.0 - m;
m = 0.0 - n_h;
if (m == 0.0) {
  if (alpha_m_im == 0.0) {
    m = alpha_m_re / br;
    alpha_m_im = 0.0;
  } else if (alpha_m_re == 0.0) {
    m = 0.0;
    alpha_m_im /= br;
  } else {
    m = alpha_m_re / br;
    alpha_m_im /= br;
  }
} else {
  n_h = std::abs(m);
  if (br > n_h) {
    a = m / br;
    n_h = br + a * m;
    m = (alpha_m_re + a * alpha_m_im) / n_h;
    alpha_m_im = (alpha_m_im - a * alpha_m_re) / n_h;
  } else if (n_h == br) {
    if (br > 0.0) {
      a = 0.5;
    } else {
      a = -0.5;
    }

    if (m > 0.0) {
      n_h = 0.5;
    } else {
      n_h = -0.5;
    }

    m = alpha_m_re * a + alpha_m_im * n_h;
    alpha_m_im = alpha_m_im * a - alpha_m_re * n_h;
  } else {
    a = br / m;
    n_h = m + a * br;
    m = (a * alpha_m_re + alpha_m_im) / n_h;
    alpha_m_im = (a * alpha_m_im - alpha_m_re) / n_h;
  }
}

mu_eff->re = 1.0 + m;
mu_eff->im = alpha_m_im;
```

```
}
```

In this example, the length of the function `CalculateAppxIndex` exceeds the defined threshold of 100.

**Correction — Refactor the Code**

One possible correction is to refactor the code so that a function perform one specific task. In this case, different tasks in the `CalculateAppxIndex` are delegated to other functions modular functions so that each functions performs one specific tasks.

```
#include <cmath>
#include <math.h>
#include <stddef.h>
#include <stdlib.h>
#define CREAL_T
typedef float real32_T;
typedef double real64_T;
typedef struct {
  real32_T re;
  real32_T im;
} creal32_T;

typedef struct {
  real64_T re;
  real64_T im;
} creal_T;
// Function Declarations
static double rt_powd_snf(double u0, double u1);
void PolarizabilityE(double&, double&, double&,double& );
void PolarizabilityM(double&, double&, double&,double& );
void Eps_eff(double&,double&,creal_T*);
void Mu_eff(double&,double&,creal_T*);
void CalculateAppxIndex(double r, double epsilon_s, double epsilon_h, double //Compliant
  lambda, double f, creal_T *eps_eff, creal_T *mu_eff)
{
  double n_h;
  double m;
  double a;
  double alpha_e_re;
  double alpha_e_im;
  double alpha_m_re;
  double alpha_m_im;
  int l;
  double br;

  //  sphere radius is 1 micron;
  //  the refractive index of inclusion
  n_h = std::sqrt(epsilon_h);

  //  the refractive index of host
  m = std::sqrt(epsilon_s) / n_h;
  n_h = 6.2831853071795862 * n_h * r / lambda;

  //  size parameter;
  a = (n_h + m) + 2.0;
  m = n_h - m;

 PolarizabilityE(a,m,alpha_e_re,alpha_e_im);
 PolarizabilityM(a,m,alpha_e_re,alpha_e_im);
 Eps_eff(alpha_e_re,alpha_e_im, eps_eff);
 Mu_eff(alpha_e_re,alpha_e_im, mu_eff);
}
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC10
**Default Threshold:** 1200

# Version History

**Introduced in R2021a**

## See Also

Check Guidelines (-guidelines)

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Number of executable lines in function body exceeds threshold

The number of executable lines in the body of a function is greater than the defined threshold

## Description

This defect is raised on a function when the number of executable lines within the body of the function is greater than the defined checker threshold. For details about how Polyspace calculates the number of executable lines in a function, see `Number of Executable Lines`

Polyspace uses the default threshold 1000 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see "Reduce Software Complexity by Using Polyspace Checkers".

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Executable Lines` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that:

- The function is overly long and performs multiple tasks.
- The function is likely to develop unexpected issues. The chance of an unexpected error increases with more lines of executable code.
- The function might contain unexpected or unplanned development.

These factors make the function difficult to maintain and debug.

### Fix

To fix this check, either refactor your code or change the checker threshold. When refactoring the code, design the functions in your code so that:

- Each function is reasonably concise.
- Each function performs one specific task.
- The functions have minimal side effects on other functions.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Overly Long Functions

This example shows how Polyspace flags functions that have more executable lines that the defines threshold of executable lines. For this example, the threshold is defined at 50. The function

CalculateAppxIndex has 116 executable line. The large number of executable lines indicate that the function is overly long. While length of a function is not a big problem by itself, overly long functions might be poorly designed, with complex data flow, many local variables, and performing many tasks. Polyspace flags the function.

```
#include <cmath>
#include <math.h>
#include <stddef.h>
#include <stdlib.h>
#define CREAL_T
typedef float real32_T;
typedef double real64_T;
typedef struct {
  real32_T re;
  real32_T im;
} creal32_T;

typedef struct {
  real64_T re;
  real64_T im;
} creal_T;
// Function Declarations
static double rt_powd_snf(double u0, double u1);

void CalculateAppxIndex(double r, double epsilon_s, double epsilon_h, double//Noncompliant
  lambda, double f, creal_T *eps_eff, creal_T *mu_eff)
{
  double n_h;
  double m;
  double a;
  double alpha_e_re;
  double alpha_e_im;
  double alpha_m_re;
  double alpha_m_im;
  int l;
  double br;

  //  sphere radius is 1 micron;
  //  the refractive index of inclusion
  n_h = std::sqrt(epsilon_h);

  //  the refractive index of host
  m = std::sqrt(epsilon_s) / n_h;
  n_h = 6.2831853071795862 * n_h * r / lambda;

  //  size parameter;
  a = (n_h + m) + 2.0;
  m = n_h - m;

  //  polarizability
  n_h = 6.2831853071795862 * rt_powd_snf(r, 3.0) / rt_powd_snf(n_h, 3.0);
  alpha_e_re = 0.0;
  alpha_e_im = 0.0;
  alpha_m_re = 0.0;
  alpha_m_im = 0.0;
  for (l = 0; l < 5; l++) {
```

```
  alpha_e_re += 0.0 * ((2.0 * (1.0 + (double)l) + 1.0) * a);
  alpha_e_im += (2.0 * (1.0 + (double)l) + 1.0) * a;
  alpha_m_re += 0.0 * ((2.0 * (1.0 + (double)l) + 1.0) * (m + 2.0));
  alpha_m_im += (2.0 * (1.0 + (double)l) + 1.0) * (m + 2.0);

  //  alpha = alpha + 1i* [(2*l+1)*(an(l) + bn(l))];
}

alpha_e_re *= n_h;
alpha_e_im *= n_h;
alpha_m_re *= n_h;
alpha_m_im *= n_h;

// alpha = aa*alpha;
n_h = f / (4.1887902047863905 * rt_powd_snf(r, 3.0));
alpha_e_re *= n_h;
alpha_e_im *= n_h;
alpha_m_re *= n_h;
alpha_m_im *= n_h;
if (alpha_e_im == 0.0) {
  m = alpha_e_re / 3.0;
  n_h = 0.0;
} else if (alpha_e_re == 0.0) {
  m = 0.0;
  n_h = alpha_e_im / 3.0;
} else {
  m = alpha_e_re / 3.0;
  n_h = alpha_e_im / 3.0;
}

br = 1.0 - m;
m = 0.0 - n_h;
if (m == 0.0) {
  if (alpha_e_im == 0.0) {
    m = alpha_e_re / br;
    alpha_e_im = 0.0;
  } else if (alpha_e_re == 0.0) {
    m = 0.0;
    alpha_e_im /= br;
  } else {
    m = alpha_e_re / br;
    alpha_e_im /= br;
  }
} else {
  n_h = std::abs(m);
  if (br > n_h) {
    a = m / br;
    n_h = br + a * m;
    m = (alpha_e_re + a * alpha_e_im) / n_h;
    alpha_e_im = (alpha_e_im - a * alpha_e_re) / n_h;
  } else if (n_h == br) {
    if (br > 0.0) {
      a = 0.5;
    } else {
      a = -0.5;
    }

    if (m > 0.0) {
      n_h = 0.5;
    } else {
      n_h = -0.5;
    }

    m = alpha_e_re * a + alpha_e_im * n_h;
    alpha_e_im = alpha_e_im * a - alpha_e_re * n_h;
  } else {
    a = br / m;
    n_h = m + a * br;
    m = (a * alpha_e_re + alpha_e_im) / n_h;
    alpha_e_im = (a * alpha_e_im - alpha_e_re) / n_h;
  }
}

eps_eff->re = epsilon_h * (1.0 + m);
eps_eff->im = epsilon_h * alpha_e_im;
if (alpha_m_im == 0.0) {
  m = alpha_m_re / 3.0;
  n_h = 0.0;
} else if (alpha_m_re == 0.0) {
  m = 0.0;
  n_h = alpha_m_im / 3.0;
} else {
  m = alpha_m_re / 3.0;
  n_h = alpha_m_im / 3.0;
}

br = 1.0 - m;
m = 0.0 - n_h;
if (m == 0.0) {
  if (alpha_m_im == 0.0) {
    m = alpha_m_re / br;
    alpha_m_im = 0.0;
  } else if (alpha_m_re == 0.0) {
    m = 0.0;
    alpha_m_im /= br;
  } else {
    m = alpha_m_re / br;
    alpha_m_im /= br;
  }
} else {
  n_h = std::abs(m);
  if (br > n_h) {
    a = m / br;
    n_h = br + a * m;
    m = (alpha_m_re + a * alpha_m_im) / n_h;
    alpha_m_im = (alpha_m_im - a * alpha_m_re) / n_h;
  } else if (n_h == br) {
    if (br > 0.0) {
      a = 0.5;
    } else {
      a = -0.5;
    }

    if (m > 0.0) {
      n_h = 0.5;
    } else {
        n_h = -0.5;
      }

      m = alpha_m_re * a + alpha_m_im * n_h;
      alpha_m_im = alpha_m_im * a - alpha_m_re * n_h;
    } else {
      a = br / m;
      n_h = m + a * br;
      m = (a * alpha_m_re + alpha_m_im) / n_h;
      alpha_m_im = (a * alpha_m_im - alpha_m_re) / n_h;
    }
  }

  mu_eff->re = 1.0 + m;
  mu_eff->im = alpha_m_im;
}
```

#### Correction — Refactor Functions

One possible correction is to refactor the function so that separate tasks are delegated to separate functions.

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC11
**Default Threshold:** 1000

# Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Number of goto statements exceeds threshold

The number of goto statements in a function is greater than the defined threshold

## Description

This defect is raised on a function when the number of goto statements in the function is greater than the define threshold. For details about how Polyspace calculates the number of goto statements, see Number of Goto Statements

Polyspace uses the default threshold zero unless you specify a threshold. To specify a selection file where you can set the threshold, use Set checkers by file (-checkers-selection-file). Also see “Reduce Software Complexity by Using Polyspace Checkers”.

When you import comments from previous analyses by using polyspace-comments-import, Polyspace copies any review information on the code metric Number of Goto Statements in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that:

- The module has an overly complicated flow of execution.
- The module might contain unexpected or unplanned development.

With multiple goto statements, it is difficult to determine the exact order of code execution in the module. This confusion might lead to bugs or an incorrect result. The complex data flow of such code makes the module difficult to maintain and debug.

### Fix

To fix this check, either change the checker threshold or refactor your code. You can replace most uses of the goto statement by more straightforward control structures. In instances where a goto statement is necessary, document it and justify the checker by using an annotation in the code. You might want to change the threshold to avoid triggering this check when checking legacy codebases.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Using goto Statements

```
#include<stdbool.h>
bool* getCondition();
void foo(){//Noncompliant
    bool* cond;
    cond = getCondition();
    while(*(cond+0)){
        //...
        while(*(cond+1)){
            //...
            while(*(cond+2)){
                //...
                if(*(cond+3)){
                goto HARDBREAK;
                }
            }
        }
    }
HARDBREAK:
        ;
        //....
        return;
}
```

In this example, a `goto` statement is used for conditionally breaking the flow of execution out of a deeply nested control structure. Polyspace flags the function because the number of `goto` statements in the function greater than the defined threshold, which is zero by default.

#### Correction — Refactor the Code or Change the Threshold

One possible correction is to refactor the code so that there are no `goto` statements in the code. For instance, you might want to use a `return` statement to get out of a deeply nested structure.

```
#include<stdbool.h>
bool* getCondition();
void foo(){//Compliant
    bool* cond;
    cond = getCondition();
    while(*(cond+0)){
        //...
        while(*(cond+1)){
            //...
            while(*(cond+2)){
                //...
                if(*(cond+3)){
                return;
                }
            }
        }
    }

}
```

Alternatively, you might consider this instance of `goto` as acceptable. In that case, change the threshold by modifying the checkers selection XML file to resolve the check.

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC13
**Default Threshold:** 0

## Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Depth of call nesting exceeds threshold

The nesting depth of control structures in a function is greater than the defined nesting depth threshold of a function

## Description

This defect is raised when the nesting depth of control structures in a function, such as `for`, `if-else`, `switch`, or `while`, exceeds the defined nesting depth threshold. For details about how Polyspace calculates the nesting depth, see `Number of Call Levels`.

Polyspace uses the default threshold 4 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see “Reduce Software Complexity by Using Polyspace Checkers”.

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Call Levels` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that:

- The function is difficult to read and understand.
- The function performs too many tasks at once.
- The function contains unexpected or unplanned development.

These factors make the module difficult to maintain and debug.

### Fix

To fix this check, either refactor your code or change the checker threshold. When refactoring your code:

- Design the function to perform a single task.
- Delegate unrelated tasks to other functions.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Functions That Have Excessive Call Nesting

```
int foo(int i, int j, int k) //Noncompliant
{
    int m=0;
    for (i=0; i<10; i++)
    {
        for (j=0; j<10; j++)
        {
            for (k=0; k<10; k++)
            {
                if (i < 2 && j>5 && k<7 )
                m += 1;
                else
                {
                    if (i > 5 && j<8 && k >8){
                        m +=-1 ;
                    }

                    else{
                        m += 2;
                    }
                }
            }
        }
    }
    return m;
}
```

In this example, the depth of call nesting is 5, which is above the default threshold of 4. The function iterates over three integers and modifies the integer `m` conditionally. The high depth of nesting makes reading and understanding the function difficult.

**Correction — Refactor the Function**

One possible correction is to delegate different tasks to different functions. For instance, delegate the conditional actions in `foo` to another function `bar`.

```
int bar(int i, int j, int k){//Compliant
int m = 0;
    if (i < 2 && j>5 && k<7 ){
        m = 1;
    }
    else
    {
        if (i > 5 && j<8 && k >8){
            m =-1 ;
        }

        else{
            m = 2;
        }
    }
return m;
}

int foo(int i, int j, int k) //Compliant
{
    int m=0;
    for (i=0; i<10; i++)
    {
        for (j=0; j<10; j++)
        {
            for (k=0; k<10; k++)
            {
                m += bar(i,j,k);
            }
        }
    }
    return m;
}
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC14
**Default Threshold:** 4

# Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Number of local static variables exceeds threshold

The number of local static variables in a function is greater than the defined threshold

## Description

This defect is raised on a function when the number of local static variables in the function is greater than the defined checker threshold. For details about how Polyspace calculates the number of local static variables in a function, see `Number of Local Static Variables`.

Polyspace uses the default threshold 20 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see "Reduce Software Complexity by Using Polyspace Checkers".

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Local Static Variables` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

The content of a static variable might be influenced by many different ways. When your function has too many static variables, it indicates that the data in your code is not properly encapsulated and your functions might have unexpected interdependencies. These factor make your code difficult to test and maintain, and might introduce bugs that are difficult to diagnose.

### Fix

To fix this check, either refactor your code or change the checker threshold. When refactoring your code:

- Encapsulate and modularize independent data or code.
- Communicate between functions by using local variables instead of global or static variables.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Functions That Have Too Many Static Variables

```
void foo(void){//Noncompliant

    static int a;
    static int b;
    static int c;
    static int d;
    static int e;
    static int f;
    static int g;
    //...
}
```

In this example, the function `foo` has seven local static variables, which is more than the specified threshold of five. Polyspace flags the function.

#### Correction — Encapsulate Related Data

One possible correction is to encapsulate related data in structures or classes. In this code, the related integer variables are encapsulated into a structure. The function `foo` now contains only one static copy of the structure..

```
typedef struct {
    int a;
    int b;
    int c;
    int d;
    int e;
    int f;
    int g;
} data_vars;
void foo(void){//Compliant

    static data_vars A;
    //...
}
```

### Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** `SC09`
**Default Threshold:** 20

## Version History

**Introduced in R2021a**

### See Also

`Check Guidelines (-guidelines)`

#### Topics

“Reduce Software Complexity by Using Polyspace Checkers”
“Modify Default Behavior of Bug Finder Checkers”

# Number of local non-static variables exceeds threshold

The number of local nonstatic variables in a function is greater than the defined threshold

## Description

This defect is raised on a function when the number of local nonstatic variables in the function is greater than the defined checker threshold. For details about how Polyspace calculates the number of local nonstatic variables in a function, see `Number of Local Non-Static Variables`.

Polyspace uses the default threshold 20 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see "Reduce Software Complexity by Using Polyspace Checkers".

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Local Non-Static Variables` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that:

- The function is overly long.
- The function performs many tasks rather than one specific task.

These factors make the module difficult to maintain and debug.

### Fix

To fix this check, either refactor your code, or change the checker threshold. When refactoring your code:

- Consider splitting a function into smaller modules that performs a specific task.
- Consider bundling variables that have a similar role into containers such as classes, structures, vectors, or maps.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Functions with Excessive Number of Local Nonstatic variables

```
#include <string>
void foo(void){//Noncompliant

    int Value1,Value2,Value3,Value4,Value5,Value6,Value7,Value8,Value9,Value10,Value11;
    std::string Name1,Name2,Name3,Name4,Name5,Name6,Name7,Name8,Name9,Name10,Name11;
    //....

}
```

In this example, The names and associated value of eleven separate entities are declared as separate variables. it is difficult to keep track of such a high number of variables. If you need to include more entities in the code, you need to declare more variables. This kind of code is difficult to test, maintain, and debug. Polyspace flags the function.

#### Correction — Bundle Related Data into Containers

One possible correction is to bundle the related data into containers. For instance, in this code the data is bundled into a `std::map` container, which is then wrapped in a class. Then in the code, you need to declare only one local variable and use the variable to store, manage, and access the data. This code is easier to maintain, test, and debug.

```
#include <string>
#include <map>
class Instance{
    public:
    Instance();
    Instance(int, std::string);
    protected:
    addData(int,std::string);
    getValue(std::string);
    //...
    private:
    std::map<std::string, int> DataMap;

};
void foo(void){//Compliant

    Instance A;
    //....
    A.addData(5,"Name");

}
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** `SC08`
**Default Threshold:** 20

# Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

“Reduce Software Complexity by Using Polyspace Checkers”
“Modify Default Behavior of Bug Finder Checkers”

# Number of function calls exceeds threshold

The number of function calls in a function is greater than the defined call occurrence threshold of a function

## Description

This defect is raised when the body of a function contains more calls to functions than the specified call occurrence threshold of a function. For details about how Polyspace calculates the number of function call occurrences, see `Number of Call Occurrences`

Polyspace uses the default threshold 50 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see “Reduce Software Complexity by Using Polyspace Checkers”.

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Call Occurrences` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that:

- The function performs too many tasks.
- The function might have high interdependency with multiple other functions.
- Changes in some other function might have unexpected impact on the flagged function.

These factors make the function difficult to maintain and debug.

### Fix

To fix this check, either refactor your code or change the checker threshold. When refactoring the code, design the functions in your code so that:

- Each function performs one specific task.
- The functions have minimal side effects on other functions.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Functions That Have Too Many Function Calls

```
long long power(double x, int n){
    long long BN = 1;
    for(int i = 0; i<n;++i){
        BN*=x;
```

```
     }
     return BN;
 }
long long factorial(int n){
     long long BN = 1;
     for(int i = 1; i<=n;++i){
         BN*=i;
     }
     return BN;
 }

 double AppxIndex(double m, double f){//Noncompliant
     double U = (power(m,2) - 1)/(power(m,2)*factorial(2));
     double V = (power(m,4) + 27*power(m,2)+38)/factorial(3)*(2*power(m,2)+3);
     return (1+2*f*power(U,2)*(1+power(m,2)*factorial(static_cast<int>(U/V))
       + power(m,3)/power(m,3)*(U-V)))/( (1-2*f*power(U,2)*(1+power(m,2)*
        factorial(static_cast<int>(U/V)) + power(m,3)/power(m,3)*(U-V)))); }
```

The function AppxIndex contains 17 different function calls, which is greater than the specified call occurrence threshold of 10. Such high call occurrence indicate that this function is doing too many tasks. It also indicates that the function has high interdependencies with other functions. These factor make AppIndex difficult to maintain, test, or debug.

**Correction — Refactor Code**

One possible correction is to refactor the function. Here, the function is split into smaller functions that performs specific tasks.

```
  long long power(double x, int n){//Compliant
    long long BN = 1;
    for(int i = 0; i<n;++i){
        BN*=x;
    }
    return BN;
}
long long factorial(int n){//Compliant
    long long BN = 1;
    for(int i = 1; i<=n;++i){
        BN*=i;
    }
    return BN;
}

double CalculateU(double m){//Compliant
    return (power(m,2) - 1)/(power(m,2)*factorial(2));
}

double CalculateV(double m){//Compliant
    return (power(m,4) + 27*power(m,2)+38)/factorial(3)*(2*power(m,2)+3);
}

double CalculateMid(double m, double f){//Compliant
    double U = CalculateU(m);
    double V = CalculateU(m);
    return (2*f*power(U,2)*(1+power(m,2)*factorial(static_cast<int>(U/V))
             + power(m,3)/power(m,3)*(U-V)));
}
```

```
double AppxIndex(double m, double f){//Compliant

    return (1+CalculateMid(m,f))/( (1-CalculateMid(m,f)));
}
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC06
**Default Threshold:** 50

# Version History

**Introduced in R2021a**

## See Also

Check Guidelines (-guidelines)

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Number of function parameters exceeds threshold

The number of arguments of a function is greater than the defined threshold

## Description

This defect is raised on a function when the number of its arguments is greater than the defined checker threshold. For details about how Polyspace calculates the number of parameters, see `Number of Function Parameters`

Polyspace uses the default threshold 5 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see "Reduce Software Complexity by Using Polyspace Checkers".

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Function Parameters` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker indicates that:

- The function might have unacceptably high degree of dependence on other functions.
- The function might be performing more than one specific task. A best practice is to delegate one specific task to one function.
- The function might contain unexpected or unplanned development.
- The function might hinder performance because registers cannot hold all parameters.

These factors make the function difficult to maintain and debug.

### Fix

To fix this check, either refactor your code or change the threshold in the checker selection XML. You might want to split the function into smaller chunks that performs a specific task and does not take more than the number of parameters specified as threshold. If the parameters of a functions are related, you might consider bundling them into structures.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Functions With Excessive Numbers of Parameters

```
#include<vector>
#include<complex>
#define PI 3.1416
std::vector<std::complex<double> >CalculateCoefficient(double, double, int);
```

```
double CalculateMfactor(double, double);
std::complex<double> CalculateEffectiveIndex(//Noncompliant
                    double Radius, double Index1, double Index2,
                     double Wavelength, double FillFactor,
                     int Cutoff){
    double RelativeIndex = Index1/Index2;
    double SizeParameter = 2*PI*Radius*Index1/Wavelength;
    std::complex<double> Polarization = 0;
    std::vector<std::complex<double>> Coefficient=
          CalculateCoefficient(RelativeIndex,SizeParameter,Cutoff);
        //...
     for (const auto& z : Coefficient){
         Polarization += z;
     }
     double Multiplier = CalculateMfactor(FillFactor,Radius);
     std::complex<double> Neff = (1.0 + Multiplier*Polarization)/
                (1.0 - Multiplier*Polarization);
     return Neff;
}
```

In this example, the function `CalculateEffectiveIndex` takes six parameters, which is greater than the defined threshold five. The high number of parameter indicate that the function might be performing more than one specific task, and it might have high degree of dependencies with other functions. Polyspace flags the function as noncompliant.

#### Correction — Refactor The Code so that Functions Perform One Specific Task

One possible correction is to refactor the function so that it performs one specific task. In this code, the function `CalculateEffectiveIndex` is refactored into two smaller functions, each of which performs one specific task. The functions are easier to debug and maintain because they take less parameters than the defined threshold. Polyspace does not flag these functions.

```
#include<vector>
#include<complex>
#define PI 3.1416
std::vector<std::complex<double>>CalculateCoefficient(double, double, int);
double CalculateMfactor(double, double);

std::complex<double> CalculatePolarization(//Compliant
        double Radius, double Index1, double Index2,
        double Wavelength, int Cutoff){
    double RelativeIndex = Index1/Index2;
    double SizeParameter = 2*PI*Radius*Index1/Wavelength;
    std::complex<double> Polarization = 0;
    std::vector<std::complex<double> > Coefficient=
              CalculateCoefficient(RelativeIndex,SizeParameter,Cutoff);
        //...
     for (const auto& z : Coefficient){
         Polarization += z;
     }

     return Polarization;
}

std::complex<double> CalculateEffectiveIndex
      (std::complex<double> Polarization, double Multiplier){//Compliant
     std::complex<double> Neff = (1.0 + Multiplier*Polarization)/
              (1.0 - Multiplier*Polarization);
     return Neff;
}
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC07
**Default Threshold:** 5

# Version History

**Introduced in R2021a**

## See Also

Check Guidelines (-guidelines)

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Number of static paths exceeds threshold

The number of static paths in a function is greater than the defined threshold

## Description

This defect is raised on a function when the number of static paths in the function is greater than the defined checker threshold. For details about how Polyspace calculates the number of static paths in a function, see `Number of Paths`

Polyspace uses the default threshold 80 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see “Reduce Software Complexity by Using Polyspace Checkers”.

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Paths` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that the function has too many possible execution paths. When there are too many execution paths, it might be difficult to test all paths exhaustively. Such functions are difficult to debug, test and maintain.

### Fix

To fix this check, either refactor your code or change the checker threshold. When refactoring your code:

- Avoid putting control structures in long sequences.
- Split a complex function into multiple functions that are simpler and easy to test.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Functions That Have High Number of Paths

```
int afunc (int x);
int foo(int x,int y) //Noncompliant
{
    int flag;
    if (x <= 0){
        if (x > 10 ) { return 0; }
    }
    if (x<-240) {
        if (x < -2565) {
            return (x < -253 ? 0: afunc (x <22566 ? 1: afunc(x < -25103 ? 0: 6)));
        }
    }
    for (int i = 0; i< 10; i++)
    {
        while (x < y ) flag = 1;
        do {++x;} while (i<7);
        flag = 0;
    }
    return flag;
}
```

In this example, the function `foo` has many branching statements points, resulting in a path number of 45, which is greater than the specified threshold of 40. Because the function has many execution paths, testing the function is difficult and testing might fail to cover all execution paths. Polyspace flags the function as noncompliant.

#### Correction — Refactor Your Code

One possible correction is to split the function into two functions.

```
int afunc (int x);
int foo2(int x,int y)//Compliant
{

    if (x <= 0){
        if (x > 10 ) { return 0; }
    }
    if (x<-240) {
        if (x < -2565) {
            return (x < -253 ? 0: afunc (x <22566 ? 1: afunc(x < -25103 ? 0: 6)));
        }
    }
}

int bar(int x,int y){//Complaint
    int flag;
    for (int i = 0; i< 10; i++)
    {
        while (x < y ) flag = 1;
        do {++x;} while (i<7);
        flag = 0;
    }
    return flag;
}
```

The functions `foo2` and `bar` have acceptable number of paths and are easier to test compared to `foo`.

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC15
**Default Threshold:** 80

## Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Number of return statements exceeds threshold

The number of `return` statements in a function is greater than the defined threshold

## Description

This defect is raised on a function when the number of `return` statement in the function is greater than the defined checker threshold. For details about how Polyspace calculates the number of return statements in a function, see `Number of Return Statements`

Polyspace uses the default threshold 1 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see "Reduce Software Complexity by Using Polyspace Checkers".

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Return Statements` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Having multiple return statements makes it difficult to determine what object a function might return. Such confusion might lead to bugs and maintenance issues that are difficult to debug.

### Fix

To fix this check, use a single return statement. For instance, store the return value in an object and define it conditionally instead of returning different objects in different conditionalized return statement.

## Examples

### Avoid Multiple `return` Statements

```
int afunc (int x);
int foo2(int x,int y)//Noncompliant
{

    if (x <= 0){
        if (x > 10 ) { return 0; }
    }
    if (x<-240) {
        if (x < -2565) {
            return (x < -253 ? 0: afunc (x <22566 ? 1: afunc(x < -25103 ? 0: 6)));
        }
    }
}
```

In this example, the `return` statement of `foo` is conditionalized, leading to two `return` statements, which exceeds the default return statement threshold of one. Polyspace flags the function `foo` as noncompliant.

#### Correction — Declare Object to Store Return value

One possible correction is to declare an object which is then conditionally defined to have the appropriate return value. This object is then returned by the function by using a single return statement.

```
int afunc (int x);
int foo2(int x,int y)//Compliant
{
    int returnData;
    if (x <= 0){
        if (x > 10 ) { returnData =  0; }
    }
    if (x<-240) {
        if (x < -2565) {
            returnData =  (x < -253 ? 0: afunc (x <22566 ? 1: afunc(x < -25103 ? 0: 6)));
        }
    }
    return returnData;
}
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC16
**Default Threshold:** 1

# Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Number of statements exceeds threshold

The number of statements in a function is greater than the defined threshold

## Description

This defect is raised on a function when the number of statements in the function is greater than the defined threshold of the checker. For details about how Polyspace calculates the number of statements in a function, see `Number of Instructions`.

Polyspace uses the default threshold 50 unless you specify a threshold for the checker. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see “Reduce Software Complexity by Using Polyspace Checkers”.

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Instructions` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that:

- The purpose of the function is unclear.
- The function performs multiple tasks.
- The function has high interdependency with other modules.
- The function contains unexpected or unplanned development.

These factors make the module difficult to maintain and debug.

### Fix

To fix this check, either refactor your code or change the checker threshold. When refactoring the code, design the functions in your code so that:

- Each function performs one specific task.
- The functions have minimal side effects on other functions.
- Independent data and code are properly isolated and encapsulated.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Overly Long Functions

```
#include <cmath>
#include <math.h>
#include <stddef.h>
```

```
#include <stdlib.h>
#define CREAL_T
typedef float real32_T;
typedef double real64_T;
typedef struct {
  real32_T re;
  real32_T im;
} creal32_T;

typedef struct {
  real64_T re;
  real64_T im;
} creal_T;
// Function Declarations
static double rt_powd_snf(double u0, double u1);

void CalculateAppxIndex(double r, double epsilon_s, double epsilon_h, double //Noncompliant
  lambda, double f, creal_T *eps_eff, creal_T *mu_eff)
{
  double n_h;
  double m;
  double a;
  double alpha_e_re;
  double alpha_e_im;
  double alpha_m_re;
  double alpha_m_im;
  int l;
  double br;

  //  sphere radius is 1 micron;
  //  the refractive index of inclusion
  n_h = std::sqrt(epsilon_h);

  //  the refractive index of host
  m = std::sqrt(epsilon_s) / n_h;
  n_h = 6.2831853071795862 * n_h * r / lambda;

  //  size parameter;
  a = (n_h + m) + 2.0;
  m = n_h - m;

  //  polarizability
  n_h = 6.2831853071795862 * rt_powd_snf(r, 3.0) / rt_powd_snf(n_h, 3.0);
  alpha_e_re = 0.0;
  alpha_e_im = 0.0;
  alpha_m_re = 0.0;
  alpha_m_im = 0.0;
  for (l = 0; l < 5; l++) {
    alpha_e_re += 0.0 * ((2.0 * (1.0 + (double)l) + 1.0) * a);
    alpha_e_im += (2.0 * (1.0 + (double)l) + 1.0) * a;
    alpha_m_re += 0.0 * ((2.0 * (1.0 + (double)l) + 1.0) * (m + 2.0));
    alpha_m_im += (2.0 * (1.0 + (double)l) + 1.0) * (m + 2.0);

    //  alpha = alpha + 1i* [(2*l+1)*(an(l) + bn(l))];
  }

  alpha_e_re *= n_h;
  alpha_e_im *= n_h;
```

```
alpha_m_re *= n_h;
alpha_m_im *= n_h;

// alpha = aa*alpha;
n_h = f / (4.1887902047863905 * rt_powd_snf(r, 3.0));
alpha_e_re *= n_h;
alpha_e_im *= n_h;
alpha_m_re *= n_h;
alpha_m_im *= n_h;
if (alpha_e_im == 0.0) {
  m = alpha_e_re / 3.0;
  n_h = 0.0;
} else if (alpha_e_re == 0.0) {
  m = 0.0;
  n_h = alpha_e_im / 3.0;
} else {
  m = alpha_e_re / 3.0;
  n_h = alpha_e_im / 3.0;
}

br = 1.0 - m;
m = 0.0 - n_h;
if (m == 0.0) {
  if (alpha_e_im == 0.0) {
    m = alpha_e_re / br;
    alpha_e_im = 0.0;
  } else if (alpha_e_re == 0.0) {
    m = 0.0;
    alpha_e_im /= br;
  } else {
    m = alpha_e_re / br;
    alpha_e_im /= br;
  }
} else {
  n_h = std::abs(m);
  if (br > n_h) {
    a = m / br;
    n_h = br + a * m;
    m = (alpha_e_re + a * alpha_e_im) / n_h;
    alpha_e_im = (alpha_e_im - a * alpha_e_re) / n_h;
  } else if (n_h == br) {
    if (br > 0.0) {
      a = 0.5;
    } else {
      a = -0.5;
    }

    if (m > 0.0) {
      n_h = 0.5;
    } else {
      n_h = -0.5;
    }

    m = alpha_e_re * a + alpha_e_im * n_h;
    alpha_e_im = alpha_e_im * a - alpha_e_re * n_h;
  } else {
    a = br / m;
    n_h = m + a * br;
```

```
      m = (a * alpha_e_re + alpha_e_im) / n_h;
      alpha_e_im = (a * alpha_e_im - alpha_e_re) / n_h;
    }
  }

  eps_eff->re = epsilon_h * (1.0 + m);
  eps_eff->im = epsilon_h * alpha_e_im;
  if (alpha_m_im == 0.0) {
    m = alpha_m_re / 3.0;
    n_h = 0.0;
  } else if (alpha_m_re == 0.0) {
    m = 0.0;
    n_h = alpha_m_im / 3.0;
  } else {
    m = alpha_m_re / 3.0;
    n_h = alpha_m_im / 3.0;
  }

  br = 1.0 - m;
  m = 0.0 - n_h;
  if (m == 0.0) {
    if (alpha_m_im == 0.0) {
      m = alpha_m_re / br;
      alpha_m_im = 0.0;
    } else if (alpha_m_re == 0.0) {
      m = 0.0;
      alpha_m_im /= br;
    } else {
      m = alpha_m_re / br;
      alpha_m_im /= br;
    }
  } else {
    n_h = std::abs(m);
    if (br > n_h) {
      a = m / br;
      n_h = br + a * m;
      m = (alpha_m_re + a * alpha_m_im) / n_h;
      alpha_m_im = (alpha_m_im - a * alpha_m_re) / n_h;
    } else if (n_h == br) {
      if (br > 0.0) {
        a = 0.5;
      } else {
        a = -0.5;
      }

      if (m > 0.0) {
        n_h = 0.5;
      } else {
        n_h = -0.5;
      }

      m = alpha_m_re * a + alpha_m_im * n_h;
      alpha_m_im = alpha_m_im * a - alpha_m_re * n_h;
    } else {
      a = br / m;
      n_h = m + a * br;
      m = (a * alpha_m_re + alpha_m_im) / n_h;
      alpha_m_im = (a * alpha_m_im - alpha_m_re) / n_h;
    }
  }

  mu_eff->re = 1.0 + m;
  mu_eff->im = alpha_m_im;
}
```

In this example, the function `CalculateAppxIndex` has more instructions than the default threshold of 50. Such high number of instructions indicate that the function has an ill-defined purpose and performs many tasks at once. The function is difficult to debug. Polyspace flags the function.

#### Correction — Delegate Independent Tasks to Functions

One possible correction is to refactor the function by delegating individual functions to smaller functions. Such modular design enables easier debugging and efficient code reuse.

```
#include <cmath>
#include <math.h>
#include <stddef.h>
#include <stdlib.h>
#define CREAL_T
typedef float real32_T;
typedef double real64_T;
typedef struct {
  real32_T re;
  real32_T im;
} creal32_T;

typedef struct {
  real64_T re;
  real64_T im;
} creal_T;
// Function Declarations
static double rt_powd_snf(double u0, double u1);
void PolarizabilityE(double&, double&, double&,double& );
void PolarizabilityM(double&, double&, double&,double& );
void Eps_eff(double&,double&,creal_T*);
void Mu_eff(double&,double&,creal_T*);
void CalculateAppxIndex(double r, double epsilon_s, double epsilon_h, double
  lambda, double f, creal_T *eps_eff, creal_T *mu_eff)//Compliant
{
  double n_h;
  double m;
  double a;
  double alpha_e_re;
  double alpha_e_im;
  double alpha_m_re;
  double alpha_m_im;
  int l;
  double br;

  //  sphere radius is 1 micron;
  //  the refractive index of inclusion
  n_h = std::sqrt(epsilon_h);

  //  the refractive index of host
  m = std::sqrt(epsilon_s) / n_h;
  n_h = 6.2831853071795862 * n_h * r / lambda;

  //  size parameter;
  a = (n_h + m) + 2.0;
  m = n_h - m;

 PolarizabilityE(a,m,alpha_e_re,alpha_e_im);
 PolarizabilityM(a,m,alpha_e_re,alpha_e_im);
 Eps_eff(alpha_e_re,alpha_e_im, eps_eff);
 Mu_eff(alpha_e_re,alpha_e_im, mu_eff);
}
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC12
**Default Threshold:** 50

# Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Number of lines exceeds threshold

The number of total lines in a file is greater than the defined threshold

## Description

This defect is raised on a file when the number of total lines in the file is greater than the defined threshold of the checker. For details about how Polyspace calculates the total number of lines in a file, see `Number of Lines`.

Polyspace uses the default threshold 10,000 unless you specify a threshold for the checker. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see “Reduce Software Complexity by Using Polyspace Checkers”.

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Number of Lines` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that:

- The file is too long.
- The file contains too many different functions.
- The file contains too many verbose comments.

These factors make the file difficult to maintain and debug.

### Fix

To fix this check, either refactor your code or change the checker threshold. When refactoring the code, design the modules of your project so that:

- Related and independent code is encapsulated and separated.
- The modules are separated based on tasks of appropriate scope.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Overdeveloped Files

```
//file1.cpp
#include <cmath>//Noncompliant
#include <math.h>
#include <stddef.h>
#include <stdlib.h>
#define CREAL_T
```

```
typedef float real32_T;
typedef double real64_T;
typedef struct {
  real32_T re;
  real32_T im;
} creal32_T;

typedef struct {
  real64_T re;
  real64_T im;
} creal_T;
// Function Declarations
static double rt_powd_snf(double u0, double u1);

void CalculateAppxIndex(double r, double epsilon_s, double epsilon_h, double
  lambda, double f, creal_T *eps_eff, creal_T *mu_eff)
{
  double n_h;
  double m;
  double a;
  double alpha_e_re;
  double alpha_e_im;
  double alpha_m_re;
  double alpha_m_im;
  int l;
  double br;

  //  sphere radius is 1 micron;
  //  the refractive index of inclusion
  n_h = std::sqrt(epsilon_h);

  //  the refractive index of host
  m = std::sqrt(epsilon_s) / n_h;
  n_h = 6.2831853071795862 * n_h * r / lambda;

  //  size parameter;
  a = (n_h + m) + 2.0;
  m = n_h - m;

  //  polarizability
  n_h = 6.2831853071795862 * rt_powd_snf(r, 3.0) / rt_powd_snf(n_h, 3.0);
  alpha_e_re = 0.0;
  alpha_e_im = 0.0;
  alpha_m_re = 0.0;
  alpha_m_im = 0.0;
  for (l = 0; l < 5; l++) {
    alpha_e_re += 0.0 * ((2.0 * (1.0 + (double)l) + 1.0) * a);
    alpha_e_im += (2.0 * (1.0 + (double)l) + 1.0) * a;
    alpha_m_re += 0.0 * ((2.0 * (1.0 + (double)l) + 1.0) * (m + 2.0));
    alpha_m_im += (2.0 * (1.0 + (double)l) + 1.0) * (m + 2.0);

    //  alpha = alpha + 1i* [(2*l+1)*(an(l) + bn(l))];
  }

  alpha_e_re *= n_h;
  alpha_e_im *= n_h;
  alpha_m_re *= n_h;
  alpha_m_im *= n_h;
```

```
// alpha = aa*alpha;
n_h = f / (4.1887902047863905 * rt_powd_snf(r, 3.0));
alpha_e_re *= n_h;
alpha_e_im *= n_h;
alpha_m_re *= n_h;
alpha_m_im *= n_h;
if (alpha_e_im == 0.0) {
  m = alpha_e_re / 3.0;
  n_h = 0.0;
} else if (alpha_e_re == 0.0) {
  m = 0.0;
  n_h = alpha_e_im / 3.0;
} else {
  m = alpha_e_re / 3.0;
  n_h = alpha_e_im / 3.0;
}

br = 1.0 - m;
m = 0.0 - n_h;
if (m == 0.0) {
  if (alpha_e_im == 0.0) {
    m = alpha_e_re / br;
    alpha_e_im = 0.0;
  } else if (alpha_e_re == 0.0) {
    m = 0.0;
    alpha_e_im /= br;
  } else {
    m = alpha_e_re / br;
    alpha_e_im /= br;
  }
} else {
  n_h = std::abs(m);
  if (br > n_h) {
    a = m / br;
    n_h = br + a * m;
    m = (alpha_e_re + a * alpha_e_im) / n_h;
    alpha_e_im = (alpha_e_im - a * alpha_e_re) / n_h;
  } else if (n_h == br) {
    if (br > 0.0) {
      a = 0.5;
    } else {
      a = -0.5;
    }

    if (m > 0.0) {
      n_h = 0.5;
    } else {
      n_h = -0.5;
    }

    m = alpha_e_re * a + alpha_e_im * n_h;
    alpha_e_im = alpha_e_im * a - alpha_e_re * n_h;
  } else {
    a = br / m;
    n_h = m + a * br;
    m = (a * alpha_e_re + alpha_e_im) / n_h;
    alpha_e_im = (a * alpha_e_im - alpha_e_re) / n_h;
  }
}

eps_eff->re = epsilon_h * (1.0 + m);
eps_eff->im = epsilon_h * alpha_e_im;
if (alpha_m_im == 0.0) {
  m = alpha_m_re / 3.0;
  n_h = 0.0;
} else if (alpha_m_re == 0.0) {
  m = 0.0;
  n_h = alpha_m_im / 3.0;
} else {
  m = alpha_m_re / 3.0;
  n_h = alpha_m_im / 3.0;
}

br = 1.0 - m;
m = 0.0 - n_h;
if (m == 0.0) {
  if (alpha_m_im == 0.0) {
    m = alpha_m_re / br;
    alpha_m_im = 0.0;
  } else if (alpha_m_re == 0.0) {
    m = 0.0;
    alpha_m_im /= br;
  } else {
    m = alpha_m_re / br;
    alpha_m_im /= br;
  }
} else {
  n_h = std::abs(m);
  if (br > n_h) {
    a = m / br;
    n_h = br + a * m;
    m = (alpha_m_re + a * alpha_m_im) / n_h;
    alpha_m_im = (alpha_m_im - a * alpha_m_re) / n_h;
  } else if (n_h == br) {
    if (br > 0.0) {
      a = 0.5;
    } else {
      a = -0.5;
    }

    if (m > 0.0) {
      n_h = 0.5;
    } else {
      n_h = -0.5;
    }

    m = alpha_m_re * a + alpha_m_im * n_h;
    alpha_m_im = alpha_m_im * a - alpha_m_re * n_h;
  } else {
    a = br / m;
    n_h = m + a * br;
    m = (a * alpha_m_re + alpha_m_im) / n_h;
    alpha_m_im = (a * alpha_m_im - alpha_m_re) / n_h;
  }
}
```

```
  mu_eff->re = 1.0 + m;
  mu_eff->im = alpha_m_im;
}
```

In this example, the file `file1.cpp` has 184 lines, which is more than the specified checker threshold of 100. This file might be overdeveloped. Polyspace raises the defect.

**Correction — Refactor Your Code**

One possible correction is to distribute the different modules of the code into different files. For instance, the different tasks in `CalculateAppxIndex` are delegated to other functions and their implementations are moved to `file2.cpp`. The type definitions and header file inclusions are moved to another header file. The file `file1.cpp` now has 40 lines, which is below the threshold.

```
//file2.cpp//Compliant
//Implementations of these functions:
void PolarizabilityE(double&, double&, double&,double& ){
//...
}
void PolarizabilityM(double&, double&, double&,double& ){
//...
}
void Eps_eff(double&,double&,creal_T*){
//...}

void Mu_eff(double&,double&,creal_T*){
//...
}

//header.h//Compliant
#include <cmath>
#include <math.h>
#include <stddef.h>
#include <stdlib.h>
#define CREAL_T
typedef float real32_T;
typedef double real64_T;
typedef struct {
  real32_T re;
  real32_T im;
} creal32_T;

typedef struct {
  real64_T re;
  real64_T im;
} creal_T;

//file1.cpp//Compliant
#include"header.h"
#include"file2.cpp"
// Function Declarations
static double rt_powd_snf(double u0, double u1);
void PolarizabilityE(double&, double&, double&,double& );
void PolarizabilityM(double&, double&, double&,double& );
void Eps_eff(double&,double&,creal_T*);
void Mu_eff(double&,double&,creal_T*);
void CalculateAppxIndex(double r, double epsilon_s,
  double epsilon_h, double
  lambda, double f, creal_T *eps_eff, creal_T *mu_eff)
{
  double n_h;
  double m;
  double a;
  double alpha_e_re;
  double alpha_e_im;
  double alpha_m_re;
  double alpha_m_im;
  int l;
  double br;

  //  sphere radius is 1 micron;
  //  the refractive index of inclusion
  n_h = std::sqrt(epsilon_h);

  //  the refractive index of host
  m = std::sqrt(epsilon_s) / n_h;
  n_h = 6.2831853071795862 * n_h * r / lambda;

  //  size parameter;
  a = (n_h + m) + 2.0;
  m = n_h - m;

 PolarizabilityE(a,m,alpha_e_re,alpha_e_im);
 PolarizabilityM(a,m,alpha_e_re,alpha_e_im);
 Eps_eff(alpha_e_re,alpha_e_im, eps_eff);
 Mu_eff(alpha_e_re,alpha_e_im, mu_eff);
}
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC01
**Default Threshold:** 10000

# Version History

**Introduced in R2021a**

## See Also

Check Guidelines (-guidelines)

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Cyclomatic complexity exceeds threshold

The cyclomatic complexity of a function is greater than the defined cyclomatic complexity threshold of a function

## Description

Polyspace calculates the cyclomatic complexity of a function by adding one to the number of decision points. A decision point is a statement that causes your program to branch into two paths. This defect is raised when the cyclomatic complexity of a function is greater than the defined cyclomatic complexity threshold. For details about how Polyspace calculates cyclomatic complexity, see `Cyclomatic Complexity`.

Polyspace uses the default threshold 10 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see “Reduce Software Complexity by Using Polyspace Checkers”.

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Cyclomatic Complexity` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that the function contains too many branches. Such functions are difficult to test and might contain unknown defects or bugs that are difficult to debug.

### Fix

To fix this check:

- Refactor your code to avoid nested control structures.
- Refactor your code to split a complex function into multiple functions that are simpler and easy to test.
- Modify the checker selection XML file to raise the cyclomatic complexity threshold.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Functions That Have High Cyclomatic Complexity

```
int afunc (int x);
int foo(int x,int y) //Noncompliant
{
    int flag;
    if (x <= 0){
        if (x > 10 ) { return 0; }
    }
    if (x<-240) {
        if (x < -2565) {
            return (x < -253 ? 0: afunc (x <22566 ? 1: afunc(x < -25103 ? 0: 6)));
        }
    }
    for (int i = 0; i< 10; i++)
    {
        while (x < y ) flag = 1;
        do {++x;} while (i<7);
        flag = 0;
    }
    return flag;
}
```

In this example, the function `foo` has too many decision points, resulting in a cyclomatic complexity of 11, which is greater than the default threshold of 10. Because the function has many decision points, testing the function is difficult and testing might fail to cover all execution paths. Polyspace flags the function as noncompliant.

#### Correction — Refactor the Function

One possible correction is to split the function into two functions.

```
int afunc (int x);
int foo2(int x,int y)//Compliant
{

    if (x <= 0){
        if (x > 10 ) { return 0; }
    }
    if (x<-240) {
        if (x < -2565) {
            return (x < -253 ? 0: afunc (x <22566 ? 1: afunc(x < -25103 ? 0: 6)));
        }
    }
}

int bar(int x,int y){//Complaint
    int flag;
    for (int i = 0; i< 10; i++)
    {
        while (x < y ) flag = 1;
        do {++x;} while (i<7);
        flag = 0;
    }
    return flag;
}
```

The functions `foo2` and `bar` have acceptable cyclomatic complexity and are easier to test compared to `foo`.

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC17
**Default Threshold:** 10

## Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

# Language scope exceeds threshold

The language scope of a function is greater than the defined threshold

## Description

Language scope represents the cost of maintaining or updating a function. For instance, if an operand occurs many times in a function, then changing the operand name is costly. The language scope of such a function would be high. This defect is raised when the language scope of a function exceeds the defined threshold. For details about how Polyspace calculates language scope, see `Language Scope`.

**Note** The language scope calculated by Polyspace is a floating point number. In this checker, the floating point language scope is converted to an integer by rounding it to the second decimal place and then multiplying it by 100.

Polyspace uses the default threshold 400 unless you specify a threshold. To specify a selection file where you can set the threshold, use `Set checkers by file (-checkers-selection-file)`. Also see “Reduce Software Complexity by Using Polyspace Checkers”.

When you import comments from previous analyses by using `polyspace-comments-import`, Polyspace copies any review information on the code metric `Language Scope` in the previous result to this checker in the current result. If the current result contains the same code metric, the review information is copied to the code metric as well.

### Risk

Violation of this checker might indicate that:

- Change to an operand might require many change in the function.
- The function might be performing multiple tasks at once.
- The function might have high degree of interdependency with other functions.

These factors make the module difficult to maintain and debug.

### Fix

To fix this check, either refactor your code or change the checker threshold. When refactoring the code, design the functions in your code so that:

- Each function performs one specific task.
- The functions have minimal side effects on other functions.

A best practice is to check the complexity of a module early in development to avoid costly post-development refactoring.

## Examples

### Avoid Functions That Have Many Recurrence of Operands

```
 long long power(double x, int n){
     long long BN = 1;
     for(int i = 0; i<n;++i){
         BN*=x;
     }
     return BN;
 }

 double AppxIndex(double m, double f){//Noncompliant
     double U = (power(m,2) - 1)/(power(m,2)+2);
     double V = (power(m,4) + 27*power(m,2)+38)/(2*power(m,2)+3);
     return (1+2*f*power(U,2)*(1+power(m,2)*U*V + power(m,3)/
                power(m,3)*(U-V)))/( (1-2*f*power(U,2)*(1+power(m,2)*U*V +
                power(m,3)/power(m,3)*(U-V))));
 }
```

In this example, In this function, the operand `power` is repeated many times. Any change to `power` requires careful examination of the function to make sure unexpected behaviors are not introduced. The language scope of the function `AppxIndex` is700, which is above the default language scope of 400. Polyspace flags the function as noncompliant

#### Correction — Refactor the Code

One possible correction is to refactor the code so that operands are not repeated many times. In this example, the function `AppxIndex` is refactored so that unrelated tasks are performed by independent functions. These functions are compliant.

```
// This code calculates effective index of materials  as decribed in
// the formula in 10.1364...
// power(x,n) returns the nth power of x (x^n)
// n is an integer
// x is a double
// return type is long long
long long power(double x, int n){//Compliant
    long long BN = 1;
    for(int i = 0; i<n;++i){
        BN*=x;
    }
    return BN;
}
// CalculateU(m) calculates the first intermediate variable
// required to calculate  polarization
// m is the relative refractive index
// return type is double;
double CalculateU(double m){//Compliant
    return (power(m,2) - 1)/(power(m,2)+2);
}
// CalculateV(m) calculates the second intermediate variable
// required to calculate  polarization
// m is the relative refractive index
// return type is double;
double CalculateV(double m){//Compliant
    return (power(m,4) + 27*power(m,2)+38)/(2*power(m,2)+3);
}
```

```
// CalculateMid(m,f) calculates the large term present in both numerator and denominator
// of the effective index calculation
// m is the relative refractive index
// f is the fillfactor
// return type is double;
double CalculateMid(double m, double f){//Compliant
    double U = CalculateU(m);
    double V = CalculateU(m);
    return 2*f*power(U,2)*(1+power(m,2)*U*V + power(m,3)/power(m,3)*(U-V));
}
//AppxIndex(m,f) calculates the approximate effective index
// m is the relative refractive index
// f is the fillfactor
//return type is double
double AppxIndex(double m, double f){//Compliant
    return (1+CalculateMid(m,f))/( (1-CalculateMid(m,f)));
}
```

## Check Information

**Group: Software Complexity**
**Language:** C | C++
**Acronym:** SC18
**Default Threshold:** 400

# Version History

**Introduced in R2021a**

## See Also

`Check Guidelines (-guidelines)`

### Topics

"Reduce Software Complexity by Using Polyspace Checkers"
"Modify Default Behavior of Bug Finder Checkers"

29

# Custom Coding Rules

# Group 1: Files

The custom rules 1.*x* in Polyspace enforce naming conventions for files and folders. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

| Number | Rule | Details |
|---|---|---|
| 1.1 | All source file names must follow the specified pattern. | Only the base name is checked. A source file is a file that is not included. |
| 1.2 | All source folder names must follow the specified pattern. | Only the folder name is checked. A source file is a file that is not included. |
| 1.3 | All include file names must follow the specified pattern. | Only the base name is checked. An include file is a file that is included. |
| 1.4 | All include folder names must follow the specified pattern. | Only the folder name is checked. An include file is a file that is included. |

# Group 2: Preprocessing

The custom rules 2.*x* in Polyspace enforce naming conventions for macros. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

| Number | Rule | Details |
|---|---|---|
| 2.1 | All macros must follow the specified pattern. | Macro names are checked before preprocessing. |
| 2.2 | All macro parameters must follow the specified pattern. | Macro parameters are checked before preprocessing. |

# Group 3: Type definitions

The custom rules 3.*x* in Polyspace enforce naming conventions for `typedef` aliases.

For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

| Number | Rule | Details |
|---|---|---|
| 3.1 | All integer types must follow the specified pattern. | Applies to integer and Boolean types specified by `typedef` statements. For example:<br>- `typedef signed long int64_t;`<br>- `typedef int64_t sLong_t;`<br>Does not apply to enumeration types. |
| 3.2 | All float types must follow the specified pattern. | Applies to double and float types specified by `typedef` statements. For example:<br>- `typedef double float64_t;`<br>- `typedef float64_t Lfloat;` |
| 3.3 | All pointer types must follow the specified pattern. | Applies to pointer types specified by `typedef` statements. For example:<br>- `typedef int* p_int;`<br>- `typedef p_int int_;` |
| 3.4 | All array types must follow the specified pattern. | Applies to array types specified by `typedef` statements. For example:<br>- `typedef int a_int_3[3];`<br>- `typedef int3_arr in3af[3];` |
| 3.5 | All function pointer types must follow the specified pattern. | Applies to function pointer types specified by `typedef` statements. For example:<br>- `typedef void (*func_fp)(int);`<br>- `typedef func_fp funcc;` |

# Group 4: Structures

The custom rules 4.*x* in Polyspace enforce naming conventions for structured data types. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

| Number | Rule | Details |
|---|---|---|
| 4.1 | All `struct` tags must follow the specified pattern. | |
| 4.2 | All `struct` types must follow the specified pattern. | `struct` types are aliases for previously defined structures (defined with the `typedef` or `using` keyword). |
| 4.3 | All `struct` fields must follow the specified pattern. | |
| 4.4 | All `struct` bit fields must follow the specified pattern. | |

# Group 5: Classes (C++)

The custom rules 5.*x* in Polyspace enforce naming conventions for classes and class members. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

| Number | Rule | Details |
|---|---|---|
| 5.1 | All class names must follow the specified pattern. | |
| 5.2 | All class types must follow the specified pattern. | Class types are aliases for previously defined classes (defined with the `typedef` or `using` keyword). |
| 5.3 | All data members must follow the specified pattern. | |
| 5.4 | All function members must follow the specified pattern. | |
| 5.5 | All static data members must follow the specified pattern. | |
| 5.6 | All static function members must follow the specified pattern. | |
| 5.7 | All bitfield members must follow the specified pattern. | |

# Group 6: Enumerations

The custom rules 6.*x* in Polyspace enforce naming conventions for enumerations. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

| Number | Rule | Details |
|---|---|---|
| 6.1 | All enumeration tags must follow the specified pattern. | |
| 6.2 | All enumeration types must follow the specified pattern. | Enumeration types are aliases for previously defined enumerations (defined with the `typedef` or `using` keyword). |
| 6.3 | All enumeration constants must follow the specified pattern. | |

# Group 7: Functions

The custom rules 7.*x* in Polyspace enforce naming conventions for functions and function parameters. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

| Number | Rule | Details |
|---|---|---|
| 7.1 | All global functions must follow the specified pattern. | A global function is a function with external linkage. |
| 7.2 | All static functions must follow the specified pattern. | A static function is a function with internal linkage. This rule does not apply to class member functions. |
| 7.3 | All function parameters must follow the specified pattern. | In C++, applies to non-member functions. |

# Group 8: Constants

The custom rules 8.*x* in Polyspace enforce naming conventions for constants. For information on how to enable these rules, see `Check custom rules (-custom-rules)`. These rules do not apply to constants that are defined within a class or to constants that are function parameters.

| Number | Rule | Details |
|---|---|---|
| 8.1 | All global nonstatic constants must follow the specified pattern. | A global nonstatic constant is a constant with external linkage. |
| 8.2 | All global static constants must follow the specified pattern. | A global static constant is a constant with internal linkage. |
| 8.3 | All local nonstatic constants must follow the specified pattern. | A local nonstatic constant is a constant without linkage. |
| 8.4 | All local static constants must follow the specified pattern. | A local static constant is a constant declared static in a function. |
| 8.5 | All floating constants must follow the specified pattern. | |

# Group 9: Variables

The custom rules 9.*x* in Polyspace enforce naming conventions for variables. For information on how to enable these rules, see `Check custom rules (-custom-rules)`. These rules do not apply to variables that are defined within a class or to variables that are function parameters.

| Number | Rule | Details |
|---|---|---|
| 9.1 | All global nonstatic variables must follow the specified pattern. | A global nonstatic variable is a variable with external linkage. |
| 9.2 | All global static variables must follow the specified pattern. | A global static variable is a variable with internal linkage. |
| 9.3 | All local nonstatic variables must follow the specified pattern. | A local nonstatic variable is a variable without linkage. |
| 9.4 | All local static variables must follow the specified pattern. | A local static variable is a variable declared static in a function. |

# Group 10: Name spaces (C++)

The custom rules 10.*x* in Polyspace enforce naming conventions for namespaces. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

| Number | Rule |
|---|---|
| 10.1 | All names spaces must follow the specified pattern. |

# Group 11: Class templates (C++)

The custom rules 11.*x* in Polyspace enforce naming conventions for class templates. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

| Number | Rule | Details |
|---|---|---|
| 11.1 | All class templates must follow the specified pattern. | |
| 11.2 | All class template parameters must follow the specified pattern. | |

# Group 12: Function templates (C++)

The custom rules 12.*x* in Polyspace enforce naming conventions for function templates. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

| Number | Rule | Details |
|---|---|---|
| 12.1 | All function templates must follow the specified pattern. | Applies to nonmember function templates. |
| 12.2 | All function template parameters must follow the specified pattern. | Applies to member and nonmember function templates. |
| 12.3 | All function template members must follow the specified pattern. | |

# Group 20: Style

The custom rules 20.*x* in Polyspace enforce coding style conventions such as number of characters per line.

For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

| Number | Rule | Details |
|---|---|---|
| 20.1 | Source line length must not exceed specified number of characters. | When configuring the checker, specify:<br>• A number for the character limit. Use the **Pattern** column on the configuration or the `pattern=` line in the custom rules text file. Polyspace raises a violation if the number of characters in a line is greater than this number.<br>• A violation message such as:<br>`Line exceeds` *`n`* `characters.`<br>Use the **Convention** column on the configuration or the `convention=` line in the custom rules xml file. |

# Polyspace Results: Code Metrics

30

# Code Metrics

# Comment Density

Ratio of number of comments to number of statements

## Description

This metric specifies the ratio of comments to statements expressed as a percentage.

Based on Hersteller Initiative Software (HIS) code complexity standard:

- Polyspace ignores these comments:
  - Comments that are included before the first statement. For instance, in this code, the first comment is ignored:

    ```
    // This function implements a hashtble
    extern void hashdef();
    ```
  - Comments that start with the source code line. For instance, this comment does not count as a comment for the metric but counts as a statement instead:

    ```
     remove(i); // Remove employee record
    ```
  - Comments that appear after the first line of a multiline comment. That is, multiline comments count as one comment. For instance, this is considered a single comment:

    ```
    // This function implements
    // regular maintenance on an internal database
    ```
- Polyspace calculates this metric after preprocessing the code. If your code contains macros, they are expanded before calculating the comment density.
- Polyspace does not count these statements:
  - Structure field and class member declarations
  - Template declarations
  - Class member function definition within the class
- A statement typically ends with a semicolon, but with some exceptions. Exceptions include semicolons in `for` loops or structure field declarations. For instance, the initialization, condition, and increment within parentheses in a `for` loop is counted as one statement. This statement counts as a single statement:

  ```
  for(i=0; i <100; i++)
  ```

  If you also declare the loop counter at initialization, that statement counts as two statements.

Polyspace does not calculate this metric for the included header files.

The recommended lower limit for this metric is 20. For better readability of your code, place at least one comment for every five statements. The value of this metric is clamped to 100. Such a value indicates that your code might contain more comment than statements.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”.

## Examples

### Comment Density Calculation

```
    struct record {
    char name[40];
    long double salary;
    int isEmployed;
};

struct record dataBase[100];

struct record fetch(void);
void remove(int);

void maintenanceRoutines() {
// This function implements
// regular maintenance on an internal database
    int i;
    struct record tempRecord;

    for(i=0; i <100; i++) {
        tempRecord = fetch(); // This function fetches a record
        // from the database
        if(tempRecord.isEmployed == 0)
            remove(i);         // Remove employee record
        //from the database
    }
}
```

In this example, the comment density is 38. The calculation is done as shown in the table:

| Code | Running Total of Comments | Running Total of Statements |
|---|---|---|
| `struct record {`<br>`    char name[40];`<br>`    long double salary;`<br>`    int isEmployed;`<br>`};` | 0 | 1 |
| `struct record dataBase[100];`<br>`struct record fetch(void);`<br>`void remove(int);` | 0 | 4 |
| `void maintenanceRoutines() {` | 0 | 4 |
| `// This function implements`<br>`// regular maintenance on an internal database` | 1 | 4 |
| `int i;`<br>`struct record tempRecord;` | 1 | 6 |
| `for(i=0; i <100; i++) {` | 1 | 6 |

| Code | Running Total of Comments | Running Total of Statements |
|---|---|---|
| `tempRecord = fetch(); // This`<br>`function fetches a record`<br>`// from the database` | 2 | 7 |
| `if(tempRecord.isEmployed == 0)`<br>`remove(i);`<br>`// Remove employee record`<br>`//from the database`<br>`}`<br>`}` | 3 | 8 |

There are three comments and eight statements. The comment density is 3/8*100 = 38.

## Metric Information

**Group**: File<br>
**Acronym**: `COMF`<br>
**HIS Metric**: Yes

## See Also

`Calculate code metrics (-code-metrics)`

# Cyclomatic Complexity

Number of linearly independent paths in function body

## Description

This metric calculates the number of decision points in a function and adds one to the total. A decision point is a statement that causes your program to branch into two paths.

The recommended upper limit for this metric is 10. If the cyclomatic complexity is high, the code is both difficult to read and can cause more orange checks. Therefore, try to limit the value of this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”.

### Computation Details

The metric calculation uses the following rules to identify decision points:

- An `if` statement is one decision point.
- The statements `for` and `while` count as one decision point, even when no condition is evaluated, for example, in infinite loops.
- Boolean combinations `(&&, ||)` do not count as decision points.
- `case` statements do not count as decision points unless they are followed by a `break` statement. For instance, this code has a cyclomatic complexity of two:

  ```
  switch(num) {
          case 0:
          case 1:
          case 2:
              break;
          case 3:
          case 4:
      }
  ```

- The calculation is done after preprocessing:
  - Macros are expanded.
  - Conditional compilation is applied. The blocks hidden by preprocessing directives are ignored.

## Examples

### Function with Nested `if` Statements

```
int foo(int x,int y)
{
    int flag;
    if (x <= 0)
        /* Decision point 1*/
        flag = 1;
```

```
    else
    {
        if (x < y )
            /* Decision point 2*/
            flag = 1;
        else if (x==y)
            /* Decision point 3*/
            flag = 0;
        else
            flag = -1;
    }
    return flag;
}
```

In this example, the cyclomatic complexity of `foo` is 4.

#### Function with ? Operator

```
int foo (int x, int y) {
    if((x <0) ||(y < 0))
        /* Decision point 1*/
        return 0;
    else
        return (x > y ? x: y);
        /* Decision point 2*/
}
```

In this example, the cyclomatic complexity of `foo` is 3. The `?` operator is the second decision point.

#### Function with `switch` Statement

```
#include <stdio.h>


int foo(int x,int y, int ch)
{
    int val = 0;
    switch(ch) {
    case 1:
        /* Decision point 1*/
        val = x + y;
        break;
    case 2:
        /* Decision point 2*/
        val = x - y;
        break;
    default:
        printf("Invalid choice.");
    }
    return val;
}
```

In this example, the cyclomatic complexity of `foo` is 3.

#### Function with Nesting of Different Control-Flow Statements

```
int foo(int x,int y, int bound)
```

```
{
    int count = 0;
    if (x <= y)
        /* Decision point 1*/
        count = 1;
    else
        while(x>y) {
            /* Decision point 2*/
            x--;
            if(count< bound) {
                /* Decision point 3*/
                count++;
            }
        }
    return count;
}
```

In this example, the cyclomatic complexity of `foo` is 4.

## Metric Information

**Group**: Function
**Acronym**: `VG`
**HIS Metric**: Yes

## See Also

`Calculate code metrics (-code-metrics)`

### Topics

"Compute Code Complexity Metrics Using Polyspace"

# Estimated Function Coupling

Measure of complexity between levels of call tree

## Description

This metric provides an approximate measure of complexity between different levels of the call tree. The metric is defined as:

*number of call occurrences – number of function definitions + 1*

If there are more function definitions than function calls, the estimated function coupling result is negative.

This metric:

- Counts function calls and function definitions in the current file only.

  It does not count function definitions in a header file included in the current file.
- Treats `static` and `inline` functions like any other function.

## Examples

### Same Function Called Multiple Times

```
void checkBounds(int *);
int getUnboundedValue();

int getBoundedValue(void) {
    int num = getUnboundedValue();
    checkBounds(&num);
    return num;
}

void main() {
    int input1=getBoundedValue(), input2= getBoundedValue(), prod;
    prod = input1 * input2;
    checkBounds(&prod);
}
```

In this example, there are:

- 5 call occurrences. Both `getBoundedValue` and `checkBounds` are called twice and `getUnboundedValue` is called once.
- 2 function definitions. `main` and `getBoundedValue` are defined.

Therefore, the Estimated function coupling is `5 - 2 + 1 = 4`.

### Negative Estimated Function Coupling

```
int foobar(int a, int b){
    return a+b;
}

int bar(int b){
    return b+2;
}

int foo(int a){
    return a<<2;
}

int main(int x){
    foobar(x,x+2);
    return 0;
}
```

This example shows how you can get a negative estimated function coupling result. In this example, you see:

- `1` function call in `main`.
- `4` defined functions: `foobar`, `bar`, `foo`, and `main`.

Therefore, the estimated function coupling is `1 - 4 + 1 = -2`.

## Metric Information

**Group**: File<br>
**Acronym**: `FCO`<br>
**HIS Metric**: No

## See Also

`Number of Call Occurrences` | `Calculate code metrics (-code-metrics)`

# Higher Estimate of Size of Local Variables

Total size of all local variables in function

## Description

**Note** Use Code Prover instead of Bug Finder for estimating the size of local variables. Support for estimating local variable size in Bug Finder will be removed in a future release. See “Version History” on page 30-13.

This metric provides a conservative estimate of the total size of local variables in a function. The metric is the sum of the following sizes in bytes:

- Size of function return value
- Sizes of function parameters
- Sizes of local variables
- Additional padding introduced for memory alignment

Your actual stack usage due to local variables can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. For instance, compilers store the address to which the execution returns following the function call. When computing this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. When computing this metric, Polyspace does not consider this hidden memory usage.
- Compilers optimize temporary variables in different ways. This metric excludes temporary variables. Only the variables that are explicitly declared by the user are considered.

However, the metric provides a reasonable estimate of the stack usage due to local variables.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric also takes into account `#pragma pack` directives in your code.

## Examples

### All Variables of Same Type

```
int flag();


int func(int param) {
  int var_1;
  int var_2;
  if (flag()) {
      int var_3;
      int var_4;
    } else {
      int var_5;
    }
}
```

In this example, assuming 4 bytes for `int`, the higher estimate of local variable size for `func`is 28. The breakup of the size is shown in this table.

| Variable | Size (in Bytes) | Running Total |
|---|---|---|
| Return value | 4 | 4 |
| Parameter `param` | 4 | 8 |
| Local variables `var_1` and `var_2` | `4+4=8` | 16 |
| Local variables defined in the `if` condition | `(4+4)+4=12`<br>The size of variables in the first branch is eight bytes. The size in the second branch is four bytes. The sum of the two branches is 12 bytes. | 28 |

No padding is introduced for memory alignment because all the variables involved have the same type.

### Variables of Different Types

```
char func(char param) {
  int var_1;
  char var_2;
  double var_3;
}
```

In this example, assuming one byte for `char`, four bytes for `int` and eight bytes for `double` and four bytes for alignment, the higher estimate of local variable size for `func` is 20. The alignment is usually the word size on your platform. In your Polyspace project, you specify the alignment through your target processor. For more information, see the Alignment column in `Target processor type (-target)`.

The breakup of the size is shown in this table.

| Variable | Size (in Bytes) | Running Total |
|---|---|---|
| Return value | 1 | 1 |
| Additional padding introduced before `param` is stored | 0<br>No memory alignment is required because the next variable `param` has the same size. | 1 |
| Parameter `param` | 1 | 2 |

| Variable | Size (in Bytes) | Running Total |
|---|---|---|
| Additional padding introduced before `var_1` is stored | 2<br>Memory must be aligned using padding because the next variable `var_1` requires four bytes. The storage must start from a memory address at a multiple of four. | 4 |
| `var_1` | 4 | 8 |
| Additional padding introduced before `var_2` is stored | 0<br>No memory alignment is required because the next variable `var_2` has smaller size. | 8 |
| `var_2` | 1 | 9 |
| Additional padding introduced before `var_3` is stored | 3<br>Memory must be aligned using padding because the next variable `var_3` has eight bytes. The storage must start from a memory address at a multiple of the alignment, four bytes. | 12 |
| `var_3` | 8 | 20 |

The rules for the amount of padding are:

- If the next variable stored has the same or smaller size, no padding is required.
- If the next variable has a greater size:
  - If the variable size is the same as or less than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of its size.
  - If the variable size is greater than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of the alignment.

### C++ Methods and Objects

```
class MySimpleClass {
  public:

    MySimpleClass() {};

    MySimpleClass(int) {};

    ~MySimpleClass() {};
};


int main() {
  MySimpleClass c;
  return 0;
}
```

In this example, the estimated local variable sizes are:

- Constructor `MySimpleClass::MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer, which is an implicit argument to the constructor. You specify the pointer size using the option `Target processor type (-target)`.
- Constructor `MySimpleClass::MySimpleClass(int)`: Eight bytes.

  The size comes from the `this` pointer and the `int` argument.
- Destructor `MySimpleClass::~MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer.
- `main()`: Five bytes.

  The size comes from the `int` return value and the size of object `c`. The minimum size of an object is the alignment that you specify using the option `Target processor type (-target)`.

### C++ Functions with Object Arguments

```
class MyClass {
  public:
    MyClass() {};
    MyClass(int) {};
    ~MyClass() {};
  private:
    int i[10];
};
void func1(const MyClass& c) {
}


void func2() {
  func1(4);
}
```

In this example, the estimated local variable size for `func2()` is 0. When `func2()` calls `func1()`, a temporary object of the class `MyClass` is created. The temporary variable is excluded from the calculation. Because there are no explicitly declared variables in the body of `func2`, the reported metric value is 0.

## Metric Information

**Group:** Function
**Acronym:** `LOCAL_VARS_MAX`
**HIS Metric**: No

# Version History

**Introduced in R2016b**

**R2022a: To be removed**
*Warns starting in R2022a*

Checking for local variable size estimates by using Bug Finder is not recommended. This workflow will be removed in a future release. To check for stack usage and local variable size estimates, use Code Prover. See `Calculate stack usage (-stack-usage)`. To see how Code Prover computes this code metric, see `Higher Estimate of Size of Local Variables`.

## See Also

`Lower Estimate of Size of Local Variables` | `Calculate code metrics (-code-metrics)` | `Calculate stack usage (-stack-usage)`

# Language Scope

Language scope

## Description

This metric measures the cost of maintaining or changing a function. It is calculated as:

`(N1 + N2)/(n1 + n2)`

Where:

- `N1` is the number of occurrences of operators.

  Other than identifiers (variable or function names) and literal constants, everything else counts as operators.
- `N2` is the number of occurrences of operands.
- `n1` is the number of distinct operators.
- `n2` is the number of distinct operands.

  The metric considers a literal constant with a suffix as different from the constant without the suffix. For instance, 0 and `0U` are considered different.

When reporting this metric, Polyspace rounds the calculated language scope to the first decimal place. Because the intent of this metric is to indicate the maintainability of a function, language scope of functions defined within local classes are not computed.

**Tip** To find `N1 + N2`, count the total number of tokens. To find `n1 + n2`, count the number of unique tokens.

The recommended upper limit for this metric is 4. For lower maintenance cost for a function, try to enforce an upper limit on this metric. For instance, if the same operand occurs many times, to change the operand name, you have to make many substitutions.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”.

## Examples

### Language Scope Calculation

```
int g(int);
int f(int i)
{
    if (i == 1)
        return i;
    else
        return i * g(i-1);
}
```

In this example:

- N1 = 19.
- N2 = 9.
- n1 = 12.

  The distinct operators are int, (, ), {, if, ==, return, else, *, -, ;, }.
- n2 = 4.

  The distinct operands are f, i, 1 and g.

The language scope of f is (19 + 9) / (12 + 4) = 1.8.

#### C++ Namespaces in Language Scope Calculation

```
namespace std {
  int func2() {
    return 123;
  }
};

namespace my_namespace {
  using namespace std;

  int func1(int a, int b) {
    return func2();
  }
};
```

In this example, the namespace std is implicitly associated with func2. The language scope computation treats func2() as std::func2(). Likewise, the computation treats func1() as my_namespace::func1().

For instance, the language scope value for func1 is 1.3. To break down this calculation:

- N1 + N2 = 20.
- n1 + n2 = 15.

  The distinct operators are int, ::, (, comma, ), {, return, ;, and }.

  The distinct operands are my_namespace, func1, a, b, std, and func2.

## Metric Information

**Group**: Function
**Acronym**: VOCF
**HIS Metric**: Yes

## See Also

Calculate code metrics (-code-metrics)

#### Topics

"Compute Code Complexity Metrics Using Polyspace"

# Lower Estimate of Size of Local Variables

Total size of local variables in function taking nested scopes into account

## Description

**Note** Use Code Prover instead of Bug Finder for estimating the size of local variables. Support for estimating local variable size in Bug Finder will be removed in a future release. See "Version History" on page 30-22.

This metric provides an optimistic estimate of the total size of local variables in a function. The metric is the sum of the following sizes in bytes:

- Size of function return value
- Sizes of function parameters
- Sizes of local variables

  Suppose that the function has variable definitions in nested scopes as follows:

  ```
  type func (type param_1, ...) {

    {
      /* Scope 1 */
      type var_1, ...;
    }
    {
      /* Scope 2 */
      type var_2, ...;
    }
  }
  ```

  The software computes the total variable size in each scope and uses whichever total is greatest. For instance, if a conditional statement has variable definitions, the software computes the total variable size in each branch, and then uses whichever total is greatest. If a nested scope itself has further nested scopes, the same process is repeated for the inner scopes.

  A variable defined in a nested scope is not visible outside the scope. Therefore, some compilers reuse stack space for variables defined in separate scopes. This metric provides a more accurate estimate of stack usage for such compilers. Otherwise, use the metric `Higher Estimate of Size of Local Variables`. This metric adds the size of all local variables, whether or not they are defined in nested scopes.
- Additional padding introduced for memory alignment

Your actual stack usage due to local variables can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When computing this metric, Polyspace does not consider these optimizations.

- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When computing this metric, Polyspace does not consider this hidden memory usage.
- Compilers optimize temporary variables in different ways. This metric excludes temporary variables. Only the variables that are explicitly declared by the user are considered.

However, the metric provides a reasonable estimate of the stack usage due to local variables.

When a function returns a `class`, `struct`, or `union` that has a copy constructor, the compiler might perform a return value optimization. Instead of returning a class object, the function might use a pointer to pass the value. This optimization might change the value of this metric.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric also takes into account `#pragma pack` directives in your code.

## Examples

### All Variables of Same Type

```
int flag();

int func(int param) {
  int var_1;
  int var_2;
  if (flag()) {
      int var_3;
      int var_4;
    } else {
      int var_5;
    }
}
```

In this example, assuming four bytes for `int`, the lower estimate of local variable size is 24. The breakup of the metric is shown in this table.

| Variable | Size (in Bytes) | Running Total |
|---|---|---|
| Return value | 4 | 4 |
| Parameter `param` | 4 | 8 |
| Local variables `var_1` and `var_2` | `4+4=8` | 16 |
| Local variables defined in the `if` condition | `max(4+4,4)= 8`<br>The size of variables in the first branch is eight bytes. The size in the second branch is four bytes. The maximum of the two branches is eight bytes. | 24 |

No padding is introduced for memory alignment because all the variables involved have the same type.

### Variables of Different Types

```
char func(char param) {
  int var_1;
  char var_2;
  double var_3;
}
```

In this example, assuming one byte for `char`, four bytes for `int`, eight bytes for `double` and four bytes for alignment, the lower estimate of local variable size is 20. The alignment is usually the word size on your platform. In your Polyspace project, you specify the alignment through your target processor. For more information, see the Alignment column in `Target processor type (-target)`.

The breakup of the size is shown in this table.

| Variable | Size (in Bytes) | Running Total |
|---|---|---|
| Return value | 1 | 1 |
| Additional padding introduced before `param` is stored | 0<br>No memory alignment is required because the next variable `param` has the same size. | 1 |
| Parameter `param` | 1 | 2 |
| Additional padding introduced before `var_1` is stored | 2<br>Memory must be aligned using padding because the next variable `var_1` requires four bytes. The storage must start from a memory address at a multiple of four. | 4 |
| `var_1` | 4 | 8 |
| Additional padding introduced before `var_2` is stored | 0<br>No memory alignment is required because the next variable `var_2` has smaller size. | 8 |
| `var_2` | 1 | 9 |
| Additional padding introduced before `var_3` is stored | 3<br>Memory must be aligned using padding because the next variable `var_3` requires eight bytes. The storage must start from a memory address at a multiple of the alignment, four bytes. | 12 |

| Variable | Size (in Bytes) | Running Total |
|---|---|---|
| var_3 | 8 | 20 |

The rules for the amount of padding are:

- If the next variable stored has the same or smaller size, no padding is required.
- If the next variable has a greater size:
  - If the variable size is the same as or less than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of its size.
  - If the variable size is greater than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of the alignment.

### C++ Methods and Objects

```
class MySimpleClass {
  public:

    MySimpleClass() {};

    MySimpleClass(int) {};

    ~MySimpleClass() {};
};


int main() {
  MySimpleClass c;
  return 0;
}
```

In this example, the estimated local variable sizes are:

- Constructor `MySimpleClass::MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer, which is an implicit argument to the constructor. You specify the pointer size using the option `Target processor type (-target)`.
- Constructor `MySimpleClass::MySimpleClass(int)`: Eight bytes.

  The size comes from the `this` pointer and the `int` argument.
- Destructor `MySimpleClass::~MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer.
- `main()`: Five bytes.

  The size comes from the `int` return value and the size of object `c`. The minimum size of an object is the alignment that you specify using the option `Target processor type (-target)`.

### C++ Functions with Object Arguments

```
class MyClass {
  public:
    MyClass() {};
    MyClass(int) {};
    ~MyClass() {};
  private:
    int i[10];
};
void func1(const MyClass& c) {
}


void func2() {
  func1(4);
}
```

In this example, the estimated local variable size for `func2()` is 0. When `func2()` calls `func1()`, a temporary object of the class `MyClass` is created. The temporary variable is excluded from the calculation. Because there are no explicitly declared variables in the body of `func2`, the reported metric value is 0.

#### Functions that Return Classes

```
class WrapperA {
private:
    int       start;
    int       end;
public:

    virtual int get_start() {
        return start;
    }
};


WrapperA aG;

WrapperA get_WrapperA_1() {
    //...
    return aG;
}


WrapperA  get_WrapperA_2() {
    //...
    WrapperA bL;
    return bL;
}


int main(){
    return 1;
}
```

In this example, the functions `get_WrapperA_1` and `get_WrapperA_2` return objects of the class `WrapperA`. You might expect the estimate for `get_WrapperA_1` to be equal to the size of `aG`. After return type optimization, `get_wrapperA_1()` contains a `WrapperA*` type pointer which points to a copy of `aG`. The lower estimate of the local variable size for `get_wrapperA_1()` is four bytes.

Similarly, after the return type optimization, `get_WrapperA_2` contains a `WrapperA` object and a `WrapperA*` pointer. The lower estimate of the local variable size for `get_wrapperA_2()` is 16 bytes.

### Metric Information

**Group:** Function
**Acronym:** `LOCAL_VARS_MIN`
**HIS Metric**: No

## Version History

**Introduced in R2016b**

#### R2022a: To be removed

*Warns starting in R2022a*

Checking for local variable size estimates by using Bug Finder is not recommended. This workflow will be removed in a future release. To check for stack usage and local variable size estimates, use Code Prover. See `Calculate stack usage (-stack-usage)`. To see how Code Prover computes this code metric, see `Lower Estimate of Size of Local Variables`.

### See Also

`Higher Estimate of Size of Local Variables` | `Calculate code metrics (-code-metrics)` | `Calculate stack usage (-stack-usage)`

# Maximum Stack Usage

Total size of local variables in function plus maximum stack usage from callees

## Description

*This metric is reported in a Code Prover analysis only.*

This metric provides a conservative estimate of the stack usage by a function. The metric is the sum of these sizes in bytes:

- `Higher Estimate of Size of Local Variables` (Polyspace Code Prover)
- Maximum value from the stack usages of the function callees. The computation uses the maximum stack usage of each callee.

  For instance, in this example, the maximum stack usage of `func` is the same as the maximum stack usage of `func1` or `func2`, *whichever is greater*.

  ```
  void func(void) {
      func1();
      func2();
  }
  ```

  If the function calls are in different branches of a conditional statement, this metric considers the branch with the greatest stack usage.

  The analysis does the stack size estimation later on when it has resolved which function calls actually occur. For instance, if a function call occurs in unreachable code, the stack size does not take the call into account. The analysis can also take into account calls through function pointers.

Your actual stack usage can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When estimating this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When estimating this metric, Polyspace does not consider this hidden memory usage.
- Compilers optimize temporary variables in different ways. This metric excludes temporary variables. Only the variables that are explicitly declared by the user are considered.

However, the metric provides a reasonable estimate of the stack usage.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric takes into account `#pragma pack` directives in your code.

## Examples

### Function with One Callee

```
double func(int);
double func2(int);
```

```
double func(int status) {
    double res = func2(status);
    return res;
}


double func2(int status) {
    double res;
    if(status == 0) {
      int temp;
      res = 0.0;
    }
    else {
      double temp;
      res = 1.0;
    }
    return res;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func2`: 32 bytes

  This value includes the sizes of its parameter (4 bytes), local variable `res` (8 bytes), local variable `temp` counted twice (4+8=12 bytes), and return value (8 bytes).

  The metric does not take into account that the first `temp` is no longer live when the second `temp` is defined.
- `func`: 52 bytes

  This value includes the sizes of its parameter, local variable `res`, and return value, a total of 20 bytes. This value includes the 32 bytes of maximum stack usage by its callee, `func2`.

#### Function with Multiple Callees

```
void func1(int);
void func2(void);



void func(int status) {
    func1(status);
    func2();
}


void func1(int status) {
    if(status == 0) {
      int val;
    }
    else {
      double val2;
    }
}
```

```
void func2(void) {
    double val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 16 bytes

  This value includes the sizes of its parameter (4 bytes) and local variables `val` and `val2` (4+8=12 bytes).
- `func2`: 8 bytes
- `func`: 20 bytes

  This value includes the sizes of its parameter (4 bytes) and the maximum of stack usages of `func1` and `func2` (16 bytes).

### Function with Multiple Callees in Different Branches

```
void func1(void);
void func2(void);


void func(int status) {
    if(status==0)
        func1();
    else
        func2();
}

void func1(void) {
    double val;
}


void func2(void) {
    int val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 8 bytes
- `func2`: 4 bytes
- `func`: 12 bytes

  This value includes the sizes of its parameter (4 bytes) and the maximum stack usage from the two branches (8 bytes).

### Functions with Variable Number of Parameters (Variadic Functions)

```
#include <stdarg.h>
```

```
void fun_vararg(int x, ...) {
  va_list ap;
  va_start(ap, x);
  int i;
  for (i=0; i<x; i++) {
    int j = va_arg(ap, int);
  }
  va_end(ap);
}



void call_fun_vararg1(void) {
  long long int l = 0;
  fun_vararg(3, 4, 5, 6, l);
}



void call_fun_vararg2(void) {
  fun_vararg(1,0);
}
```

In this function, `fun_vararg` is a function with variable number of parameters. The maximum stack usage of `fun_vararg` takes into account the call to `fun_vararg` with the maximum number of arguments. The call with the maximum number of arguments is the call in `call_fun_vararg1` with five arguments (one for the fixed parameter and four for the variable parameters). The maximum stack usages are:

- `fun_vararg`: 36 bytes.

  This value takes into account:

  - The size of the fixed parameter `x` (4 bytes).
  - The sizes of the variable parameters from the call with the maximum number of parameters. In that call, there are four variable arguments: three `int` and one `long long int` variable (3 times 4 + 1 times 8 = 20 bytes).
  - The sizes of the local variables `i`, `j` and `ap` (12 bytes). The size of the `va_list` variable uses the pointer size defined in the target (in this case, 4 bytes).
- `call_fun_vararg1`: 44 bytes.

  This value takes into account:

  - The stack size usage of `fun_vararg` with five arguments (36 bytes).
  - The size of local variable `l` (8 bytes).
- `call_fun_vararg2`: 20 bytes.

  Since `call_fun_vararg2` has no local variables, this value is the same as the stack size usage of `fun_vararg` with two arguments (20 bytes, of which 12 bytes are for the local variables and 8 bytes are for the two parameters of `fun_vararg`).

## Metric Information

**Group:** Function

**Acronym:** `MAX_STACK`
**HIS Metric**: No

## Version History

**Introduced in R2017b**

## See Also

`Minimum Stack Usage` (Polyspace Code Prover) | `Program Maximum Stack Usage` (Polyspace Code Prover) | `Higher Estimate of Size of Local Variables` (Polyspace Code Prover) | `Calculate code metrics (-code-metrics)` | `Calculate stack usage (-stack-usage)`

### Topics

"Determination of Program Stack Usage" (Polyspace Code Prover)
"Migrate Code Prover Workflows for Checking Coding Standards and Code Metrics to Bug Finder" (Polyspace Code Prover)

# Minimum Stack Usage

Total size of local variables in function taking nested scopes into account plus maximum stack usage from callees

## Description

*This metric is reported in a Code Prover analysis only.*

This metric provides an optimistic estimate of the stack usage by a function. Unlike the metric `Maximum Stack Usage` (Polyspace Code Prover), this metric takes nested scopes into account. For instance, if variables are defined in two mutually exclusive branches of a conditional statement, the metric considers that the stack space allocated to the variables in one branch can be reused in the other branch.

The metric is the sum of these sizes in bytes:

- `Lower Estimate of Size of Local Variables` (Polyspace Code Prover).
- Maximum value from the stack usages of the function callees. The computation uses the minimum stack usage of each callee.

  For instance, in this example, the minimum stack usage of `func` is the same as the minimum stack usage of `func1` or `func2`, *whichever is greater*.

  ```
  void func(void) {
      func1();
      func2();
  }
  ```

  If the function calls are in different branches of a conditional statement, this metric considers the branch with the least stack usage.

  The analysis does the stack size estimation later on when it has resolved which function calls actually occur. For instance, if a function call occurs in unreachable code, the stack size does not take the call into account. The analysis can also take into account calls through function pointers.

Your actual stack usage can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When estimating this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When estimating this metric, Polyspace does not consider this hidden memory usage.
- Compilers optimize temporary variables in different ways. This metric excludes temporary variables. Only the variables that are explicitly declared by the user are considered.

However, the metric provides a reasonable estimate of the stack usage.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric takes into account `#pragma pack` directives in your code.

## Examples

### Function with One Callee

```
double func2(int);


double func(int status) {
    double res = func2(status);
    return res;
}


double func2(int status) {
    double res;
    if(status == 0) {
      int temp;
      res = 0.0;
    }
   else {
      double temp;
      res = 1.0;
    }
   return res;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the minimum stack usages are:

- `func2`: 28 bytes

  This value includes the sizes of its parameter (4 bytes), local variable `res` (8 bytes), one of the two local variables `temp` (8 bytes), and return value (8 bytes).

  The metric takes into account that the first `temp` is no longer live when the second `temp` is defined. It uses the variable `temp` with data type `double` because its size is greater.
- `func`: 48 bytes

  This value includes the sizes of its parameter, local variable `res`, and return value, a total of 20 bytes. This value includes the 28 bytes of minimum stack usage by its callee, `func2`.

### Function with Multiple Callees

```
void func1(int);
void func2(void);



void func(int status) {
    func1(status);
    func2();
}


void func1(int status) {
    if(status == 0) {
      int val;
    }
    else {
      double val2;
    }
}


void func2(void) {
    double val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the minimum stack usages are:

- `func1`: 12 bytes

  This value includes the sizes of its parameter (4 bytes) and the larger of the two local variables, in this case, `val2` (8 bytes).
- `func2`: 8 bytes
- `func`: 16 bytes

  This value includes the sizes of its parameter (4 bytes) and the maximum of stack usages of `func1` and `func2` (12 bytes).

### Function with Multiple Callees in Different Branches

```
void func1(void);
void func2(void);



void func(int status) {
    if(status==0)
        func1();
    else
        func2();
}


void func1(void) {
    double val;
}


void func2(void) {
    int val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the minimum stack usages are:

- `func1`: 8 bytes
- `func2`: 4 bytes
- `func`: 8 bytes

This value includes the sizes of its parameter (4 bytes) and the minimum stack usage from the two branches (4 bytes).

### Functions with Variable Number of Parameters (Variadic Functions)

```
#include <stdarg.h>


void fun_vararg(int x, ...) {
  va_list ap;
  va_start(ap, x);
  int i;
  for (i=0; i<x; i++) {
    int j = va_arg(ap, int);
  }
  va_end(ap);
}



void call_fun_vararg1(void) {
  long long int l = 0;
  fun_vararg(3, 4, 5, 6, l);
}



void call_fun_vararg2(void) {
  fun_vararg(1,0);
}
```

In this function, `fun_vararg` is a function with variable number of parameters. The minimum stack usage of `fun_vararg` takes into account the call to `fun_vararg` with the minimum number of arguments. The call with the minimum number of arguments is the call in `call_fun_vararg2` with two arguments (one for the fixed parameter and one for the variable parameter). The minimum stack usages are:

- `fun_vararg`: 20 bytes.

  This value takes into account:

  - The size of the fixed parameter `x` (4 bytes).
  - The sizes of the variable parameters from the call with the minimum number of parameters. In that call, there is only one variable argument of type `int` (4 bytes).
  - The sizes of the local variables `i`, `j` and `ap` (12 bytes). The size of the `va_list` variable uses the pointer size defined in the target (in this case, 4 bytes).
- `call_fun_vararg1`: 44 bytes.

  This value takes into account:

  - The stack size usage of `fun_vararg` with five arguments (36 bytes, of which 12 bytes are for the local variable sizes and 20 bytes are for the fixed and variable parameters of `fun_vararg`).
  - The size of local variable `l` (8 bytes).
- `call_fun_vararg2`: 20 bytes.

Since `call_fun_vararg2` has no local variables, this value is the same as the stack size usage of `fun_vararg` with two arguments (20 bytes).

### Metric Information

**Group:** Function
**Acronym:** `MIN_STACK`
**HIS Metric**: No

## Version History

**Introduced in R2017b**

### See Also

`Program Minimum Stack Usage` (Polyspace Code Prover) | `Lower Estimate of Size of Local Variables` (Polyspace Code Prover) | `Maximum Stack Usage` (Polyspace Code Prover) | `Calculate code metrics (-code-metrics)` | `Calculate stack usage (-stack-usage)`

#### Topics

"Determination of Program Stack Usage" (Polyspace Code Prover)

# Number of Call Levels

Maximum depth of nesting of control flow structures

## Description

This metric specifies the maximum nesting depth of control flow statements such as `if`, `switch`, `for`, or `while` in a function. A function without control-flow statements has a call level 1.

The recommended upper limit for this metric is 4. For better readability of your code, try to enforce an upper limit for this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”.

## Examples

### Function with Nested `if` Statements

```
int foo(int x,int y)
{
    int flag = 0;
    if (x <= 0)
        /* Call level 1*/
        flag = 1;
    else
    {
        if (x <= y )
            /* Call level 2*/
            flag = 1;
        else
            flag = -1;
    }
    return flag;
}
```

In this example, the number of call levels of `foo` is 2.

### Function with Nesting of Different Control-Flow Statements

```
int foo(int x,int y, int bound)
{
    int count = 0;
    if (x <= y)
        /* Call level 1*/
        count = 1;
    else
        while(x>y) {
            /* Call level 2*/
            x--;
            if(count< bound) {
                /* Call level 3*/
                count++;
            }
        }
    return count;
}
```

In this example, the number of call levels of `foo` is 3.

## Metric Information

**Group**: Function
**Acronym**: `LEVEL`
**HIS Metric**: Yes

## See Also

`Calculate code metrics (-code-metrics)`

### Topics

“Compute Code Complexity Metrics Using Polyspace”

# Number of Call Occurrences

Number of calls in function body

## Description

This metric specifies the number of function calls in the body of a function.

This metric includes:

- Calls in unreachable code.
- Calls to standard library functions.
- Calls to constructors and destructors.

The metric excludes:

- Calls through a function pointer.
- Calls to virtual functions.
- Calls to `assert`. Polyspace considers `assert` as a macro and not a function.
- Inlined functions. Compilers might inline certain function calls, such as the default constructors and destructors.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”. To enforce violation of code metrics limits as Bug Finder defects, use the Software Complexity checkers. See “Reduce Software Complexity by Using Polyspace Checkers”

## Examples

### Same Function Called Multiple Times

```
int func1(void);
int func2(void);


int foo() {
    return (func1() + func1()*func1() + 2*func2());
}
```

In this example, the number of call occurrences in `foo` is 4.

### Function Called in a Loop

```
#include<stdio.h>
int getVal(void);


void fillArraySize10(int *arr) {
    for(int i=0; i<10; i++)
        arr[i]=getVal();
}

int getVal(void) {
    int val;
    printf("Enter a value:");
    scanf("%d", &val);
    return val;
}
```

In this example, the number of call occurrences in `fillArraySize10` is 1.

#### Recursive Function

```
#include <stdio.h>
int fibonacci(int);

void main() {
 int count;
 printf("How many numbers ?");
 scanf("%d",&count);
 fibonacci(count);
}

int fibonacci(int num)
{
   if ( num == 0 )
      return 0;
   else if ( num == 1 )
      return 1;
   else
      return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of call occurrences in `fibonacci` is 2.

#### Constructor Function

```
#include<iostream>
class A{
    public:
    A(){
        std::cout<<"Create A\n";
    }
    ~A() = default;
    A(const A&)=default;
    A(A&&) = default;
    virtual void bar(){ std::cout<<"A";}
};
class B: public A{
    public:
    B() = default;
     void bar() override {std::cout<<"B";}
};

void func(A& a){
    a.bar();
}

int main(){
    A obj;
    A obj2 = obj;
    B objB;
    func(obj);
    return 0;
}
```

In this example, the number of call occurances in `main` is three:

1. The constructor of class `A` in `A obj;`. This user defined constructor counts as a function call.
2. The constructor of class `B` in `B objB;`. Because the constructor of the base class `A` is user-defined, the constructor of `B` counts as a function call even though `B::B()` is declared as `=default`.
3. The call to function `func`.

The class `A` uses the default or implicit copy constructor. The call to the copy constructor in `A obj2 = obj;` does not count as a function call.

## Metric Information

**Group**: Function
**Acronym**: `NCALLS`
**HIS Metric**: No

## See Also

`Number of Called Functions` | `Calculate code metrics (-code-metrics)`

# Number of Called Functions

Number of distinct functions called within the body of a function

## Description

This metric specifies the number of distinct functions that are called by a function within its body.

This metric includes:

- Calls in unreachable code.
- Calls to standard library functions.
- Calls to user defined constructors and destructors.

The metric excludes:

- Calls through a function pointer.
- Calls to virtual functions.
- Calls to `assert`. Polyspace considers `assert` as a macro and not a function.
- Inlined functions. Compilers might inline certain function calls, such as the default or implicit constructors and destructors.

The recommended upper limit for this metric is 7. For more self-contained code, try to enforce an upper limit on this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”. To enforce violation of code metrics limits as Bug Finder defects, use the Software Complexity checkers. See “Reduce Software Complexity by Using Polyspace Checkers”

## Examples

### Same Function Called Multiple Times

```
int func1(void);
int func2(void);

int foo() {
    return (func1() + func1()*func1() + 2*func2());
}
```

In this example, the number of called functions in `foo` is 2. The called functions are `func1` and `func2`.

### Recursive Function

```
#include <stdio.h>
int fibonacci(int);

void main() {
 int count;
 printf("How many numbers ?");
 scanf("%d",&count);
 fibonacci(count);
}

int fibonacci(int num)
{
   if ( num == 0 )
      return 0;
   else if ( num == 1 )
      return 1;
   else
      return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of called functions in `fibonacci` is 1. The called function is `fibonacci` itself.

### Constructor Function

```
            #include<iostream>
class A{
    public:
    A(){
        std::cout<<"Create A\n";
    }
    ~A() = default;
    A(const A&)=default;
    A(A&&) = default;
    virtual void bar(){ std::cout<<"A";}
};
class B: public A{
    public:
    B() = default;
     void bar() override {std::cout<<"B";}
};

void func(A& a){
    a.bar();
}

int main(){
    A obj;
    A obj2 = obj;
    B objB;
    func(obj);
    return 0;
}
```

In this example, the number of called function in `main` is three:

1. The constructor of class `A`. This user defined constructor counts as a function call.
2. The constructor of class `B`. Because the constructor of the base class `A` is user-defined, the constructor of `B` counts as a function call even though `B::B()` is declared as `=default`.
3. The function `func`.

The class `A` uses the default or implicit copy constructor. The call to the copy constructor in `A obj2 = obj;` does not count as a function call.

## Metric Information

**Group**: Function
**Acronym**: `CALLS`
**HIS Metric**: Yes

## See Also

`Number of Call Occurrences` | `Number of Calling Functions` | `Calculate code metrics (-code-metrics)`

### Topics

"Compute Code Complexity Metrics Using Polyspace"

# Number of Calling Functions

Number of distinct callers of a function

## Description

This metric measures the number of distinct callers of a function.

In C++ , Polyspace does not calculate this metric for virtual functions and compiler generated implicit functions, such as default constructors and destructors. The metric is calculated for user-defined constructors and destructors. In a class hierarchy, if a base class has user-defined constructors, Polyspace counts this metric for corresponding constructors of the derived classes.

The recommended upper limit for this metric is 5. For more self-contained code, try to enforce an upper limit on this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”.

### Computation Details

Note that the metric:

- Takes into account direct callers only.
- Does not consider calls through a function pointer.
- Takes into account all function calls, including ones in unreachable code.

  However, if a caller calls a function more than once, the caller is counted only once when this metric is calculated.

## Examples

### Same Function Calling a Function Multiple Times

```
#include <stdio.h>


    int getVal() {
    int myVal;
    printf("Enter a value:");
    scanf("%d", &myVal);
    return myVal;
}

int func() {
    int val=getVal();
    if(val<0)
        return 0;
    else
        return val;
}

int func2() {
    int val=getVal();
    while(val<0)
        val=getVal();
    return val;
}
```

In this example, the number of calling functions for `getVal` is 2. The calling functions are `func` and `func2`.

#### Recursive Function

```
#include <stdio.h>




    int fibonacci(int num)
{
   if ( num == 0 )
      return 0;
   else if ( num == 1 )
      return 1;
   else
      return ( fibonacci(num-1) + fibonacci(num-2) );
}

void main() {
 int count;
 printf("How many numbers ?");
 scanf("%d",&count);
 fibonacci(count);
}
```

In this example, the number of calling functions for `fibonacci` is 2. The calling functions are `main` and `fibonacci` itself.

#### Constructor Function

```

                #include<iostream>
class A{
    public:

    A(){
        std::cout<<"Create A\n";
    }
    ~A() = default;
    A(const A&)=default;
    A(A&&) = default;
    virtual void bar(){ std::cout<<"A";}
};
class B: public A{
    public:
    B() = default;
     void bar() override {std::cout<<"B";}
};

void func(A& a){
    a.bar();
}
int main(){
    A obj;
    A obj2 = obj;
    B objB;
    func(obj);
    return 0;
}
```

In this example:

- The number of calling functions for `A::A` is two. `A::A` is called once to create `obj` and again to create `objB`. Similarly, the number of calling function for `B::` is one.
- Because both `A::bar` and `B::bar` are virtual functions, Polyspace does not calculate their number of calling functions.
- The number of calling function for `func` is one.

## Metric Information

**Group**: Function
**Acronym**: `CALLING`
**HIS Metric**: Yes

## See Also

`Number of Called Functions` | `Calculate code metrics (-code-metrics)`

### Topics

"Compute Code Complexity Metrics Using Polyspace"

# Number of Direct Recursions

Number of instances of a function calling itself directly

## Description

This metric specifies the number of direct recursions in your project.

A direct recursion is a recursion where a function calls itself in its own body. If indirect recursions do not occur, the number of direct recursions is equal to the number of recursive functions.

The recommended upper limit for this metric is 0. To avoid the possibility of exceeding available stack space, do not use recursions in your code. To detect use of recursions, check for violations of `MISRA C:2012 Rule 17.2`.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”.

## Examples

### Direct Recursion

```
int getVal(void);
int sum(int val) {
    if(val<0)
        return 0;
    else
        return (val + sum(val-1));
}

void main() {
    int count = getVal(), total;
    assert(count > 0 && count <100);
    total = sum(count);
}
```

In this example, the number of direct recursions is 1.

## Metric Information

**Group**: Project
**Acronym**: `AP_CG_DIRECT_CYCLE`
**HIS Metric**: Yes

## See Also

`MISRA C:2012 Rule 17.2` | `Calculate code metrics (-code-metrics)`

### Topics

“Compute Code Complexity Metrics Using Polyspace”

# Number of Executable Lines

Number of executable lines in function body

## Description

This metric measures the number of executable lines in a function body. When calculating the value of this metric, Polyspace excludes declarations, comments, blank lines, braces or preprocessing directives.

If the function body contains a `#include` directive, the included file source code is also calculated as part of this metric.

This metric is not calculated for C++ templates and its value is reported as `Not computed`.

## Examples

### Function with Declarations, Braces and Comments

```
void func(int, double);

    int getSign(int arg) {//Excluded
    int sign; //Excluded
    static int siNumber = 0; //Excluded
    double dNumber = 5;//Excluded
    if(arg<0) {
        sign=-1;
        func(-arg,dNumber);
        ++siNumber;
        /* func takes positive first argument */   //Excluded
    }//Excluded
    else if(arg==0)
    sign=0;
    else {
        sign=1;
        func(arg,dNumber);
        ++siNumber;
    }//Excluded
    return sign;
}//Excluded
```

In this example, the number of executable lines of `getSign` is 11. The calculation excludes:

- The definition of the function.
- The variable declarations.
- The comment `/* ... */`.
- The lines with braces only.

## Metric Information

**Group**: Function
**Acronym**: FXLN

**HIS Metric**: No

## See Also

Number of Lines Within Body | Number of Instructions | Calculate code metrics (-code-metrics)

# Number of Files

Number of source files

## Description

This metric calculates the number of source files in your project.

## Examples

### Source File Calling Function in Included File

```
#include<iostream>
#include"pow.cpp"

double power(double, double);
//Function to calculate approximate index
 double AppxIndex(double m, double f){
     double U = (power(m,2) - 1)/(power(m,2)+2); //First term
     double V = (power(m,4) + 27*power(m,2)+38)/
     (2*power(m,2)+3);// Second term
     return (1+2*f*power(U,2)*(1+power(m,2)*U*V +
      power(m,3)/power(m,3)*(U-V)))
     /( (1-2*f*power(U,2)*(1+power(m,2)*U*V
     + power(m,3)/power(m,3)*(U-V))));
 }


 int main(){
     return 0;
 }
```

In this example, the code calls the function `power`, which is defined in an included source file `pow.cpp`. When analysing the code, Polyspace does not consider the included source file. Because the included source file is not considered in the analysis, Polyspace evaluates the number of source file to be one.

## Metric Information

**Group**: Project
**Acronym**: `FILES`
**HIS Metric**: No

## See Also

`Number of Header Files` | `Calculate code metrics (-code-metrics)`

# Number of Function Parameters

Number of function arguments

## Description

This metric measures the number of function arguments.

If ellipsis is used to denote variable number of arguments, when calculating this metric, the ellipsis is not counted.

The recommended upper limit for this metric is 5. For less dependency between functions and fewer side effects, try to enforce an upper limit on this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”.

## Examples

### Function with Fixed Arguments

```
int initializeArray(int* arr, int size) {
}
```

In this example, `initializeArray` has two parameters.

### Function with Type Definition in Arguments

```
int getValueInLoc(struct {int* arr; int size;}myArray, int loc) {
}
```

In this example, `getValueInLoc` has two parameters.

### Function with Variable Arguments

```
double average ( int num, ... )
{
    va_list arg;
    double sum = 0;

    va_start ( arg, num );

    for ( int x = 0; x < num; x++ )
    {
        sum += va_arg ( arg, double );
    }
    va_end ( arg);

    return sum / num;
}
```

In this example, `average` has one parameter. The ellipsis denoting variable number of arguments is not counted.

## Metric Information

**Group**: Function
**Acronym**: `PARAM`
**HIS Metric**: Yes

## See Also

`Calculate code metrics (-code-metrics)`

### Topics

"Compute Code Complexity Metrics Using Polyspace"

# Number of Goto Statements

Number of `goto` statements

## Description

This metric measures the number of `goto` statements in a function.

`break` and `continue` statements are not counted.

The recommended upper limit on this metric is 0. For better readability of your code, avoid `goto` statements in your code. To detect use of `goto` statements, check for violations of `MISRA C:2012 Rule 15.1`.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”.

## Examples

### Function with goto Statements

```
#define SIZE 10
int initialize(int **arr, int loc);
void printString(int *);
void printErrorMessage(void);
void printExecutionMessage(void);


int main()
{
    int *arrayOfStrings[SIZE],len[SIZE],i;
    for ( i = 0; i < SIZE; i++ )
    {
        len[i] = initialize(arrayOfStrings,i);
    }

    for ( i = 0; i < SIZE; i++ )
    {
        if(len[i] == 0)
            goto emptyString;
        else
            goto nonEmptyString;
        loop: printExecutionMessage();
    }

emptyString:
    printErrorMessage();
    goto loop;
nonEmptyString:
    printString(arrayOfStrings[i]);
    goto loop;
}
```

In this example, the function `main` has 4 `goto` statements.

## Metric Information

**Group**: Function
**Acronym**: `GOTO`
**HIS Metric**: Yes

## See Also

`Calculate code metrics (-code-metrics)`

### Topics

"Compute Code Complexity Metrics Using Polyspace"

# Number of Header Files

Number of included header files

## Description

This metric measures the number of header files in the project that is considered in an analysis. Both directly and indirectly included header files are counted.

The metric gives a slightly different number than the actual number of header files that you use because Polyspace® internal header files and header files included by those files are also counted. For the same reason, the metric can vary slightly even if you do not explicitly include new header files or remove inclusion of header files from your code. For instance, the number of Polyspace® internal header files can vary if you change your analysis options.

## Examples

### Header Files Not Considered in Analysis

```
#include<iostream>
#include<string>
// FUnction to calculate power
 long long power(double x, int n){
     long long BN = 1;// long long
     for(int i = 0; i<n;++i){
         BN*=x;
     }
     return BN;
 }
```

In this example, the code does not call any of the functions defined in the `iostream` and `string`. These headers are not considered in the analysis. Polyspace calculates this metric to be zero.

## Metric Information

**Group**: Project
**Acronym**: `INCLUDES`
**HIS Metric**: No

## See Also

`Number of Files` | `Calculate code metrics (-code-metrics)`

# Number of Instructions

Number of instructions per function

## Description

This metric measures the number of instructions in a function body.

The recommended upper limit for this metric is 50. For more modular code, try to enforce an upper limit for this metric.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”.

### Computation Details

The metric is calculated using the following rules:

- A simple statement ending with a `;` is one instruction.

  If the statement is empty, it does not count as an instruction.
- A variable declaration counts as one instruction if either of these conditions are true:
  - The variable is nonstatic and it is initialized. For instance:

    ```
    int var = 0;// Single statement
    ```

    Declarations of static variables not counted as statements. These might be initialized in compile time in C++
  - The variable is initialized by a function call, such as a constructor. For instance:

    ```
    class A{
    public:
        A();
        A(int in){
            /*..*/
        }
    };
    //...
    A obj(5);//Calls A::A(int). This is considered a single statement.
    ```
- Control flow statements such as `if`, `for`, `break`, `goto`, `return`, `switch`, `while`, `do-while` count as one instruction.
- The following do not count as instructions by themselves:
  - Beginning of a block of code

    For instance, the following counts as one instruction:

    ```
    {
        int var = 1;
    }
    ```
  - Labels

For instance, the following counts as two instructions. The `case` labels do not count as instructions.

```
switch (1) {  // Instruction 1: switch
    case 0:
    case 1:
    case 2:
    default:
    break;    // Instruction 2: break
 }
```

## Examples

### Calculation of Number of Instructions

```
int func(int* arr, int size) {
    int i, countPos=0, countNeg=0, countZero = 0;
    for(i=0; i<size; i++) {
        if(arr[i] >0)
            countPos++;
        else if(arr[i] ==0)
            countZero++;
        else
            countNeg++;
    }
}
```

In this example, the number of instructions in `func` is 9. The instructions are:

1. `countPos=0`
2. `countNeg=0`
3. `countZero=0`
4. `for(i=0;i<size;i++) { ... }`
5. `if(arr[i] >=0)`
6. `countPos++`
7. `else if(arr[i]==0)`

   The ending `else` is counted as part of the `if-else` instruction.
8. `countZero++`
9. `countNeg++`

**Note** This metric is different from the number of executable lines. For instance:

- `for(i=0;i<size;i++)` has 1 instruction and 1 executable line.
- The following code has 1 instruction but 3 executable lines.

```
for(i=0;
    i<size;
    i++)
```

## Metric Information

**Group**: Function
**Acronym**: STMT
**HIS Metric**: Yes

## See Also

`Calculate code metrics (-code-metrics)`

### Topics

"Compute Code Complexity Metrics Using Polyspace"

# Number of Lines

Total number of lines in a file

## Description

This metric calculates the number of lines in a file. When calculating the value of this metric, Polyspace includes comments and blank lines.

This metric is calculated for source files and header files in the same folders as source files. If you want:

- The metric reported for other header files, change the default value of the option `Generate results for sources and (-generate-results-for)`.
- The metric not reported for header files at all, change the value of the option `Do not generate results for (-do-not-generate-results-for)` to `all-headers`.

## Examples

### Code Containing Comments and Blank Lines

```
// Function to calculate power
 long long power(double x, int n){
     long long BN = 1;// long long
     for(int i = 0; i<n;++i){
         BN*=x;
     }
     return BN;
 }
//Function to calculate approximate index
 double AppxIndex(double m, double f){
     double U = (power(m,2) - 1)/(power(m,2)+2); //First term
     double V = (power(m,4) + 27*power(m,2)+38)/
     (2*power(m,2)+3);// Second term
     return (1+2*f*power(U,2)*(1+power(m,2)*U*V +
      power(m,3)/power(m,3)*(U-V)))
     /( (1-2*f*power(U,2)*(1+power(m,2)*U*V
     + power(m,3)/power(m,3)*(U-V))));
 }
```

Because Polyspace includes comments and blank lines when calculating this metric, the total number of line in this file is 18.

## Metric Information

**Group**: File
**Acronym**: `TOTAL_LINES`
**HIS Metric**: No

## See Also

Number of Lines Without Comment | Calculate code metrics (-code-metrics)

# Number of Lines Within Body

Number of lines in function body

## Description

This metric calculates the number of lines in a function body. When calculating the value of this metric, Polyspace includes declarations, comments, blank lines, braces, and preprocessing directives. The initialization list of a member function is considered to be within the function body when calculating this metric.

If the function body contains a `#include` directive, the included file source code is also calculated as part of this metric.

This metric is not calculated for C++ templates and its value is reported as `Not computed`.

## Examples

### Function with Declarations, Braces, and Comments

```
void func(int);


int getSign(int arg) {
    int sign;
    if(arg<0) {
        sign=-1;
        func(-arg);
        /* func takes positive arguments */
    }
    else if(arg==0)
        sign=0;
    else {
        sign=1;
        func(arg);
    }
    return sign;
}
```

In this example, the number of executable lines of `getSign` is 13. The calculation includes:

- The declaration `int sign;`.
- The comment `/* ... */`.
- The two lines with braces only.

### Member Function with Initialization List

```
class A {
    bool val;
    char c;
    int i;
public:
    A();
    void foo();
};

void A::foo() {}

A::A(): val(false),
        c('A'),
        i(42)
{}
```

In this example, the function `A::foo()` has no executable lines. The value of the metric for this function is zero. The member function `A::A()` has three initializations in an initialization list. Polyspace considers these initializations to be within the function body. Because the function does not have an explicit return statement, the compiler generates an implicit return statement. Including the implicit return and the initializations, the value of this metric for `A::A()` is 4.

## Metric Information

**Group**: Function
**Acronym**: `FLIN`
**HIS Metric**: No

## See Also

`Number of Executable Lines` | `Calculate code metrics (-code-metrics)`

# Number of Lines Without Comment

Number of lines of code excluding lines that are comments or blank

## Description

This metric calculates the number of executable lines in a file. When calculating the value of this metric, Polyspace excludes lines that are comments or blank.

This metric is calculated for source files and header files in the same folders as source files. If you want:

- The metric reported for other header files, change the default value of the option `Generate results for sources and (-generate-results-for)`.
- The metric not reported for header files at all, change the value of the option `Do not generate results for (-do-not-generate-results-for)` to `all-headers`.

## Examples

### Inline Comments

```
// Function to calculate power
 long long power(double x, int n){
     long long BN = 1;// long long
     for(int i = 0; i<n;++i){
         BN*=x;
     }
     return BN;
 }
//Function to calculate approximate index
 double AppxIndex(double m, double f){
     double U = (power(m,2) - 1)/(power(m,2)+2); //First term
     double V = (power(m,4) + 27*power(m,2)+38)/
     (2*power(m,2)+3);// Second term
     return (1+2*f*power(U,2)*(1+power(m,2)*U*V +
      power(m,3)/power(m,3)*(U-V)))
     /( (1-2*f*power(U,2)*(1+power(m,2)*U*V
     + power(m,3)/power(m,3)*(U-V))));
 }
```

In this example, Polyspace calculates the number of lines in the file that are neither comment nor blank lines. Ignoring lines that are comments, such as the line `// FUnction to calculate power`, and blank lines, Polyspace evaluates the metric as `16`. When evaluating this metric, the lines of code that contain inline comments are counted as lines of code.

## Metric Information

**Group**: File
**Acronym**: `LINES_WITHOUT_CMT`
**HIS Metric**: No

## See Also

`Number of Lines` | `Calculate code metrics (-code-metrics)`

# Number of Local Non-Static Variables

Total number of local variables in function

## Description

This metric provides the number of declared local variables in a function.

The metric excludes static variables and temporary variables. To find number of static variables, use the metric `Number of Local Static Variables`.

## Examples

### Non-Structured Variables

```
int flag();


int func(int param) {
  int var_1;
  int var_2;
  if (flag()) {
      int var_3;
      int var_4;
    } else {
      int var_5;
    }
}
```

In this example, the number of local non-static variables in `func` is 5. The number does not include the function arguments and return value.

### Arrays and Structured Variables

```
typedef struct myStruct{
   char  arr1[50];
   char  arr2[50];
   int   val;
} myStruct;


void func(void) {
  myStruct var;
  char localArr[50];
}
```

In this example, the number of local non-static variables in `func` is 2: the structured variable `var` and the array `localArr`.

### Variables in Class Methods

```
class Rectangle {
    int width, height;
```

```
  public:
    void set (int,int);
    int area (void);
} rect;

int Rectangle::area (void) {
    int temp;
    temp = width * height;
    return(temp);
}
```

In this example, the number of local non-static variables in `Rectangle::area` is 1: the variable `temp`.

### Metric Information

**Group:** Function
**Acronym:** `LOCAL_VARS`
**HIS Metric**: No

## Version History

**Introduced in R2017a**

### See Also

`Number of Local Static Variables` | `Higher Estimate of Size of Local Variables` | `Lower Estimate of Size of Local Variables` | `Calculate code metrics (-code-metrics)`

# Number of Local Static Variables

Total number of local static variables in function

## Description

This metric provides the number of local static variables in a function.

## Examples

### Number of Static Variables

```
void func(void) {
  static int var_1 = 0;
  int var_2;
}
```

In this example, the number of static variables in `func` is 1. For examples of different types of variables, see `Number of Local Non-Static Variables`.

## Metric Information

**Group:** Function
**Acronym:** `LOCAL_STATIC_VARS`
**HIS Metric**: No

# Version History

**Introduced in R2017a**

## See Also

`Higher Estimate of Size of Local Variables` | `Number of Local Non-Static Variables` | `Calculate code metrics (-code-metrics)`

# Number of Paths

Estimated static path count

## Description

This metric measures the number of static paths in a function.

The recommended upper limit for this metric is 80. If the number of paths is high, the code is difficult to read and can cause more orange checks. Try to limit the value of this metric.

To enforce limits on metrics, see "Compute Code Complexity Metrics Using Polyspace".

### Computation Details

The number of paths is calculated according to these rules:

- If the statements in a function do not break the control flow, the number of paths is one.

  Even an empty statement such as `;` or empty block such as `{}` counts as one path.
- A control flow statement introduces branches and adds to the original one path.

  - `if-else if-else`: Each `if` keyword introduces a new branch. The contribution from an `if-else if-else` block is the number of branches plus one (the original path). If a catch-all `else` is present, all paths go through the block; otherwise, one path bypasses the block.

    For instance, a function with an `if(..) {} else if(..) {} else {}` statement has three paths. A function with one `if() {}` only has two paths, one that goes through the `if` block and one that bypasses the block.
  - `switch-case`: Each `case` label introduces a new branch. The contribution from a `switch` block is the number of `case` labels plus one (the original path). If a catch-all `default` is present, all paths go through the block; otherwise, one path bypasses the block.

    For instance, a function with a statement `switch (var) { case 1: .. break; case 2: .. break; default: .. }` has three paths, all going through the `switch` block. If you omit the `default`, the function still has three paths, two going through the `switch` block and one bypassing the block.
  - `for` and `while`: Each loop statement introduces a new branch. The contribution from a loop is two - a path that goes through the loop and a path that bypasses the loop.
  - `do-while`: Each `do-while` statement introduces a new branch except when the condition of the `while` statement is explicitly false. Statements written as `do{/*..*/}while(0)` do not function as loops. Such statements are often used for enclosing multiple lines of macros within braces. For instance, this `do-while` statement serves to encapsulate the multiline macro rather than create a new path:

    ```
    #define myfunc(x) do{ ...\\
                          ...\\
                          ...}while(0);
    ```

    Polyspace considers such statements to be a single path.

  Note that a statement with a ternary operator such as

```
result = a > b ? a : b;
```

is not considered as a statement that breaks the control flow.

- If more than one control flow statement are present in a sequence without any nesting, the number of paths is the product of the contributions from each control flow statement.

  For instance, if a function has three `for` loops and two `if-else` blocks, one after another, the number of paths is 2 × 2 × 2 × 2 × 2 = 32.

  If many control flow statements are present in a function, the number of paths can be large. Nested control flow statements reduce the number of paths at the cost of increasing the depth of nesting. For an example, see “Function with Nested Control Flow Statements” on page 30-67.
- The software displays specific values in cases where the metric is not calculated:
  - If `goto` statements are present in the body of the function, Polyspace cannot calculate the number of paths and shows the number of paths as `Not Computed` instead.
  - If the number of paths reaches an upper limit of 1,000,000,000, Polyspace stops the calculation and displays just the upper limit. The actual value might be higher.
- The number of path is calculated statically. This statically calculated number represents the upper limit of the possible path in a function. Typically, the number of possible path during run time is lower than the number Polyspace reports.
- Polyspace ignores compiler generated conditional statements in C++ when calculating the number of paths. The compiler might generate implicit conditional statements in operations such as initializing static variables or calling virtual functions.

## Examples

### Function with One Path

```
int func(int ch) {
    return (ch * 2);
}
```

In this example, `func` has one path.

### Function with Control Flow Statement Causing Multiple Paths

```
void func(int ch) {
    switch (ch)
    {
    case 1:
        break;
    case 2:
        break;
    case 3:
        break;
    case 4:
        break;
    default:
    }
}
```

In this example, `func` has five paths. Apart from the path that goes through the `default`, each `case` label followed by a statement causes the creation of a new path.

### Function with Nested Control Flow Statements

```
void func()
{
    int i = 0, j = 0, k = 0;
    for (i=0; i<10; i++)
    {
        for (j=0; j<10; j++)
        {
            for (k=0; k<10; k++)
            {
                if (i < 2 )
                    ;
                else
                {
                    if (i > 5)
                        ;
                    else
                        ;
                }
            }
        }
    }
}
```

In this example, `func` has six paths: three from the `for` statements, two from the `if` statements plus the original path that bypasses all control flow statements.

### Function with Control Flow Statements in Sequence

```
int look_up(int a, int b){
    if(a==b){
        return 1;
    }
    if(a>b+2){
        return 2;
    }
    if((a+b) < 10){
        return 3;
    }
    if((a+b) > 11){
        return 4;
    }
    if((a+b) >20 && (b-a)!=0){
        return 3;
    }
}
```

In this example, there are five `if` statements in sequence. Each `if` statements introduces two possible execution path. The total number of path is `2 × 2 × 2 × 2 × 2 = 32`. Control statements in sequence increases the number of path quickly. If the number of path is higher than acceptable, consider refactoring your function.

## Metric Information

**Group**: Function
**Acronym**: PATH
**HIS Metric**: Yes

## See Also

`Calculate code metrics (-code-metrics)`

### Topics

“Compute Code Complexity Metrics Using Polyspace”

# Number of Potentially Unprotected Shared Variables

Number of unprotected shared variables

## Description

This metric measures the number of variables with the following properties:

- The variable is used in more than one task.
- At least one operation on the variable is not protected from interruption by operations in other tasks.

## Examples

### Unprotected Shared Variables

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        reset();
        inc();
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}
```

In this example, Polyspace calculates the number of potentially unprotected shared variable to be one. The shared variabel `shared_var` is unprotected if you specify `task` and `interrupt_handler` as entry points and do not specify protection mechanisms.

The operation `shared_var = INT_MAX` can interrupt the other operations on `shared_var` and cause unpredictable behavior.

### Metric Information

**Group**: Project<br>
**Acronym**: `UNPSHV`<br>
**HIS Metric**: No

## Version History

**Introduced in R2018b**

### See Also

`Calculate code metrics (-code-metrics)`

# Number of Protected Shared Variables

Number of protected shared variables

## Description

This metric measures the number of variables with the following properties:

- The variable is used in more than one task.
- All operations on the variable are protected from interruption through critical sections or temporal exclusions.

## Examples

### Shared Variables Protected Through Temporal Exclusion

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        reset();
        inc();
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}
```

In this example, `shared_var` is a protected shared variable if you specify the following options:

| Option | Value |
|---|---|
| Tasks (-entry-points) | task<br>interrupt_handler |
| Temporally exclusive tasks (-temporal-exclusions-file) | temporal_exclusion.txt |

In the command line, specify these options to run the example:

```
-entry-points task,interrupt_handler  -temporal-exclusions-file temporal_exclusion.txt
```

`temporal_exclusion.txt` is a text file containing the temporally exclusive tasks separated by a space in a single line:

```
task interrupt_handler
```

The variable is shared between `task` and `interrupt_handler`. However, because `task` and `interrupt_handler` are temporally exclusive, operations on the variable cannot interrupt each other.

#### Shared Variables Protected Through Critical Sections

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void take_semaphore(void);
void give_semaphore(void);

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        reset();
        inc();
        inc();
        give_semaphore();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        interrupt();
        give_semaphore();
    }
}

void main() {
}
```

In this example, `shared_var` is a protected shared variable if you specify the following:

| Option | Value | |
|---|---|---|
| **Entry points** | `task`<br>`interrupt_handler` | |
| **Critical section details** | **Starting routine** | **Ending routine** |
| | `take_semaphore` | `give_semaphore` |

In the command line, specify these options to run the example:

```
-critical-section-begin  take_semaphore:cs1
-critical-section-end give_semaphore:cs1
-entry-points task,interrupt_handle
```

The variable is shared between `task` and `interrupt_handler`. However, because operations on the variable are between calls to the starting and ending procedure of the same critical section, they cannot interrupt each other.

### Metric Information

**Group**: Project
**Acronym**: PSHV
**HIS Metric**: No

## Version History

**Introduced in R2018b**

### See Also

`Tasks (-entry-points)` | `Critical section details (-critical-section-begin -critical-section-end)` | `Temporally exclusive tasks (-temporal-exclusions-file)` | `Calculate code metrics (-code-metrics)`

# Number of Recursions

Number of call graph cycles over one or more functions

## Description

The metric provides a quantitative estimate of the number of recursion cycles in your project. The metric is the sum of:

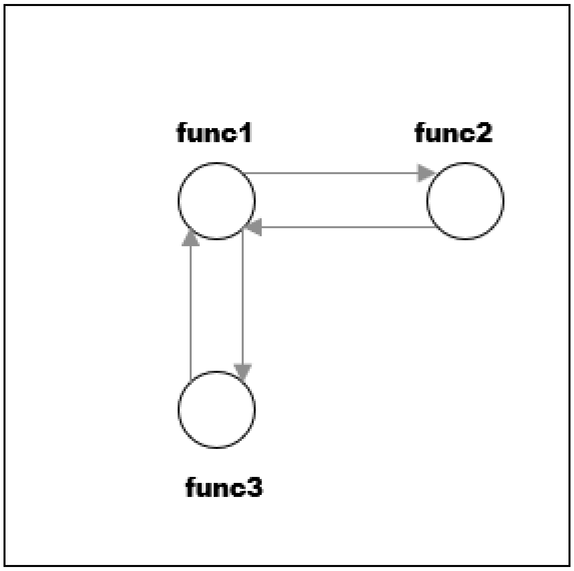
- Number of direct recursions (self recursive functions or functions calling themselves).
- Number of strongly connected components formed by the indirect recursion cycles in your project. If you consider the recursion cycles as a directed graph, the graph is strongly connected if there is a path between all pairs of vertices.

  To compute the number of strongly connected components:

  1. Draw the recursion cycles in your code.

     For instance, the recursion cycles in this example are shown below.

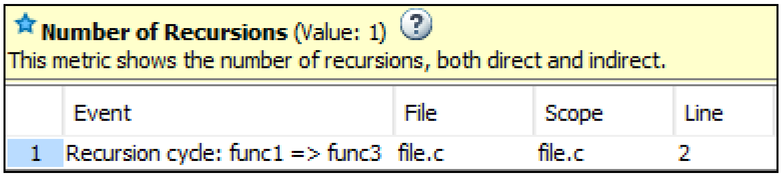
     ```
     volatile int checkStatus;
     void func1() {
        if(checkStatus) {
             func2();
        }
        else {
             func3();
        }
     }

     func2() {
        func1();
     }

     func3() {
        func1();
     }
     ```



2 Identify the number of strongly connected components formed by the recursion cycles.

In the preceding example, there is one strongly connected component. You can move from any vertex to another vertex by following the paths in the graph.

The event list below the metric shows one of the recursion cycles in the strongly connected component.



**Number of Recursions** (Value: 1)
This metric shows the number of recursions, both direct and indirect.

| | Event | File | Scope | Line |
|---|---|---|---|---|
| 1 | Recursion cycle: func1 => func3 | file.c | file.c | 2 |

Calls through a function pointer are not considered.

The recommended upper limit for this metric is 0. To avoid the possibility of exceeding available stack space, do not use recursions in your code. Recursions can tend to exhaust stack space easily. See examples of stack size growth with recursions described for this CERT-C rule that forbids recursions.

To detect use of recursions, check for violations of one of `MISRA C:2012 Rule 17.2`,MISRA C: 2004 Rule 16.2, `MISRA C++:2008 Rule 7-5-4` or JSF Rule 119. Note that:

- The rule checkers report each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.
- The rule checkers consider explicit function calls only. For instance, in C++ code, the rule checkers ignore implicit calls to constructors during object creation. However, the metrics computation considers both implicit and explicit calls.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”.

## Examples

### Direct Recursion

```
int getVal(void);
int sum(int val) {
    if(val<0)
        return 0;
    else
        return (val + sum(val-1));
}

void main() {
    int count = getVal(), total;
    assert(count > 0 && count <100);
    total = sum(count);
}
```

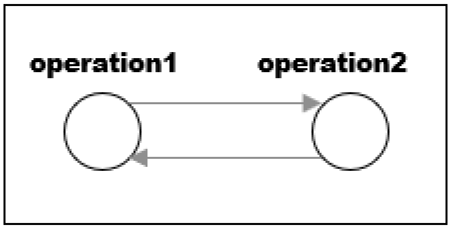
In this example, the number of recursions is 1.

A direct recursion is a recursion where a function calls itself in its own body. For direct recursions, the number of recursions is equal to the number of recursive functions.

### Indirect Recursion with One Call Graph Cycle

```
volatile int signal;
void operation2(void);

void operation1(void) {
    int stop = signal%2;
    if(!stop)
        operation2();
}

void operation2(void) {
    operation1();
}

void main() {
    operation1();
}
```

In this example, the number of recursions is one. The two functions `operation1` and `operation2` are involved in the call graph cycle `operation1` → `operation2` → `operation1`.



An indirect function is a recursion where a function calls itself through other functions. For indirect recursions, the number of recursions can be different from the number of recursive functions.

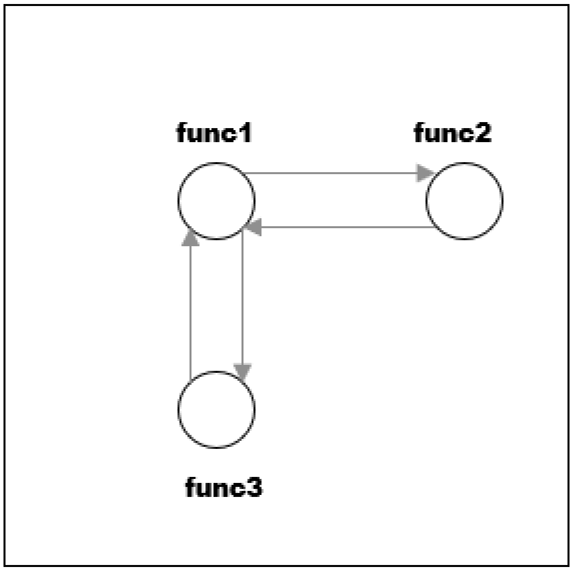
### Multiple Call Graph Cycles Forming One Strongly Connected Component

```
volatile int checkStatus;
void func1() {
   if(checkStatus) {
        func2();
   }
   else {
        func3();
   }
}

func2() {
   func1();
}

func3() {
   func1();
}
```

In this example, there are two call graph cycles:

- `func1` → `func2` → `func1`
- `func1` → `func3` → `func1`

However, the cycles form one strongly connected component. You can move from any vertex to another vertex by following the paths in the graph. Hence, the number of recursions is one.



### Indirect Recursion with Two Call Graph Cycles

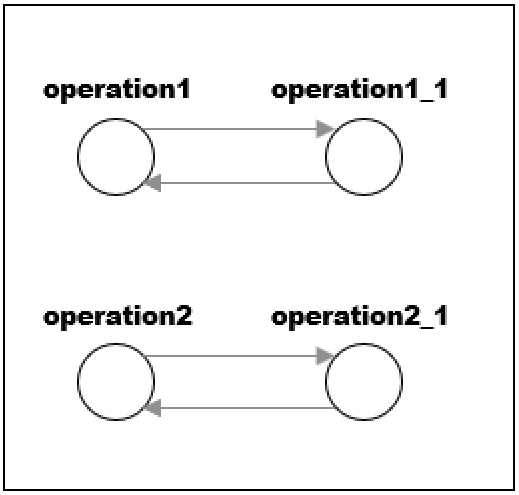
```
volatile int signal;
void operation1_1();
void operation2_1();
void operation1() {
    int stop = signal%2;
    if(!stop)
        operation1_1();
}


void operation1_1() {
    operation1();
}

void operation2() {
    int stop = signal%2;
    if(!stop)
        operation2_1();
}

void operation2_1() {
    operation2();
}

void main(){
    operation1();
    operation2();
}
```

In this example, the number of recursions is two.

There are two call graph cycles:

- `operation1` → `operation1_1` → `operation1`
- `operation2` → `operation2_1` → `operation2`

The call graph cycles form two strongly connected components.

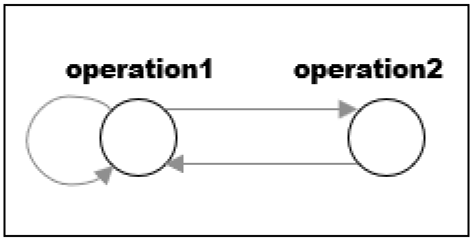


**Same Function Called in Direct and Indirect Recursion**

```
volatile int signal;
void operation2();
void operation1() {
    int stop = signal%3;
    if(stop==1)
        operation1();
    else if(stop==2)
        operation2();
}

void operation2() {
    operation1();
}

void main() {
    operation1();
}
```

In this example, the number of recursions is two:

- The strongly connected component formed by the cycle `operation1` → `operation2` → `operation1`.
- The self-recursive function `operation1`.



## Metric Information

**Group**: Project
**Acronym**: `AP_CG_CYCLE`
**HIS Metric**: Yes

## See Also

`MISRA C:2012 Rule 17.2` | `Calculate code metrics (-code-metrics)`

### Topics

"Compute Code Complexity Metrics Using Polyspace"

# Number of Return Statements

Number of `return` statements in a function

## Description

This metric measures the number of `return` statements in a function.

The recommended upper limit for this metric is 1. If one return statement is present, when reading the code, you can easily identify what the function returns.

To enforce limits on metrics, see “Compute Code Complexity Metrics Using Polyspace”.

## Examples

### Function with Return Points

```
int getSign (int arg) {
    if(arg <0)
        return -1;
    else if(arg > 0)
        return 1;
    return 0;
}
```

In this example, `getSign` has 3 `return` statements.

## Metric Information

**Group**: Function
**Acronym**: `RETURN`
**HIS Metric**: Yes

## See Also

`Calculate code metrics (-code-metrics)`

### Topics

“Compute Code Complexity Metrics Using Polyspace”

# Program Maximum Stack Usage

Maximum stack usage in the analyzed program

## Description

*This metric is reported in a Code Prover analysis only.*

This metric shows the maximum stack usage from your program.

The metric shows the maximum stack usage for the function with the highest stack usage. If you provide a complete application, the function with the highest stack usage is typically the `main` function because the `main` function is at the top of the call hierarchy. In multitasking or multithreaded applications, the program maximum stack usage is the maximum stack usage of the task or thread with the highest stack usage. For a description of maximum stack usage for a function, see the metric `Maximum Stack Usage` (Polyspace Code Prover).

## Metric Information

**Group:** Project
**Acronym:** `PROG_MAX_STACK`
**HIS Metric**: No

# Version History

**Introduced in R2017b**

## See Also

`Higher Estimate of Size of Local Variables` (Polyspace Code Prover) | `Maximum Stack Usage` (Polyspace Code Prover) | `Program Minimum Stack Usage` (Polyspace Code Prover) | `Calculate code metrics (-code-metrics)` | `Calculate stack usage (-stack-usage)`

### Topics

“Determination of Program Stack Usage” (Polyspace Code Prover)

# Program Minimum Stack Usage

Maximum stack usage in the analyzed program taking nested scopes into account

## Description

*This metric is reported in a Code Prover analysis only.*

This metric shows the maximum stack usage from your program, taking nested scopes into account. The metric provides a lower bound to the stack usage since unlike the `Program Maximum Stack Usage` (Polyspace Code Prover), the metric takes into account possible stack space reuse within a function and does not simply add the sizes of all local variables in a function.

The metric shows the minimum stack usage for the function with the highest stack usage. If you provide a complete application, the function with the highest stack usage is typically the `main` function because the `main` function is at the top of the call hierarchy. In multitasking or multithreaded applications, the program minimum stack usage is the minimum stack usage of the task or thread with the highest stack usage. For a description of minimum stack usage for a function, see the metric `Minimum Stack Usage` (Polyspace Code Prover).

Considering nested scopes is useful for compilers that reuse stack space for variables defined in nested scopes. For instance, in this code, the space for `var_1` is reused for `var_2`.

```
type func (type param_1, ...) {

  {
    /* Scope 1 */
    type var_1, ...;
  }
  {
    /* Scope 2 */
    type var_2, ...;
  }
}
```

## Metric Information

**Group:** Project
**Acronym:** `PROG_MIN_STACK`
**HIS Metric**: No

# Version History

**Introduced in R2017b**

## See Also

`Lower Estimate of Size of Local Variables` (Polyspace Code Prover) | `Minimum Stack Usage` (Polyspace Code Prover) | `Program Maximum Stack Usage` (Polyspace Code Prover) | `Calculate code metrics (-code-metrics)` | `Calculate stack usage (-stack-usage)`

**Topics**

"Determination of Program Stack Usage" (Polyspace Code Prover)

# Polyspace Reports Components

31

# Report Components

# Acronym Definitions

Create table of Polyspace acronyms used in report and their full forms

## Description

This component creates a table containing the acronyms used in the report and their full forms. Acronyms are used for Polyspace checks and result status.

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Call Hierarchy

Create table showing call graph in source code

## Description

This component creates a table showing the call hierarchy in your source code. For each function call in your source code, the table displays the following information:

- Level of call hierarchy, where the function is called.

  Each level is denoted by |. If a function call appears in the table as `|||-> `*`file_name.function_name`*, the function call occurs at the third level of the hierarchy. Beginning from `main` or an entry point, there are three function calls leading to the current call.
- File containing the function call.

  In Code Prover, the line and column is also displayed.
- File containing the function definition.

  In Code Prover, the line and column where the function definition begins is also displayed.

In addition, the table also displays uncalled functions.

This table captures the information available on the **Call Hierarchy** pane in the Polyspace user interface.

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Code and Verification Information

Create table of verification times and code characteristics

## Description

This component creates tables containing verification times and code characteristics such as number of lines.

## Properties

### Include Verification Time Information

If you select this option, the report contains verification times broken down by phase.

- For Polyspace Bug Finder, the phases are `compilation`, `pass0`, `pass1`, etc.
- For Polyspace Code Prover, the phases are `compilation`, `global`, `function`, etc.

### Include Code Details

If you select this option, the report contains the following code characteristics:

- Number of files
- Number of lines
- Number of lines without comment

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Code Metrics Details

Create table of Polyspace metrics broken down by file and function

## Description

This component creates a table containing metrics from a Polyspace project. Polyspace calculates various code metrics about the project, source files, and individual functions. For a list of these code metrics, see “Code Metrics”.

## See Also

**Topics**
“Customize Existing Bug Finder Report Template”

# Code Metrics Summary

Create table of Polyspace metrics

## Description

This component creates a table containing metrics from a Polyspace project. The metrics are the same as those displayed under `Code Metrics Details`. However, the file and function metrics are not broken down by individual files and functions. Instead, the table provides the minimum and maximum value of a file metric over all files and a function metric over all functions.

## See Also

### Topics

"Customize Existing Bug Finder Report Template"

# Code Verification Summary

Create table of Polyspace analysis results

## Description

This component creates tables containing the following results:

- Number of results
- Number of coding rule violations for each coding rule type such as MISRA C
- Number of defects, for Polyspace Bug Finder results
- Number of checks of each color, for Polyspace Code Prover results
- Whether the project passed or failed the software quality objective

## Properties

### Include Checks from Polyspace Standard Library Stub Functions

Unless you deselect this option, the tables contain Polyspace Code Prover checks that appear in Polyspace stubs for the standard library functions.

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Coding Rules Details

Create table of coding rule violations broken down by file

## Description

This component creates tables containing coding rule violations broken down by each file in the Polyspace project. For each rule violation, the table contains the following information:

- Rule number
- Rule description
- Function containing the violation
- (Code Prover only) Line and column number
- Review information such as classification, status and comments

## Properties

### Select Coding Rules Type

Using this option, you can choose which coding rule violations to display. You can display violations for the following set of coding rules:

- MISRA C rules
- MISRA AC AGC rules
- MISRA C++ rules
- JSF C++ rules
- Custom coding rules

### Display by

Using this option, you can break down the display of coding rule violations by file.

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Coding Rules Summary

Create table with number of coding rule violations

## Description

This component creates a table containing the number of coding rule violations. You can choose whether to break this information down by rule number or file.

## Properties

### Select Coding Rules Type

Using this option, you can choose which coding rule violations to display. You can display violations for the following set of coding rules:

- MISRA C rules
- MISRA AC AGC rules
- MISRA C++ rules
- JSF C++ rules
- Custom coding rules

### Include Files/Rules with No Problems Detected

If you select this option, the table displays:

- Files that do not contain coding rule violations
- Rules that your code does not violate

### Display by

Using this option, you can break down the display of coding rule violations by:

- Rule number
- File

## See Also

### Topics
"Customize Existing Bug Finder Report Template"

# Configuration Parameters

Create table of analysis options, assumptions and coding rules configuration

## Description

This component creates the following tables:

- *Polyspace settings*: The analysis options that you used to obtain your results. The table lists command-line version of the options along with their values.
- *Analysis assumptions*: The assumptions used to obtain your Code Prover results. The table lists only the modifiable assumptions. For assumptions that you cannot change, see the Polyspace documentation.
- *Coding rules configuration*: The coding rules whose violations you checked for. The table lists the rule number, rule description and other information about the rules.
- *Files with compilation errors*: If your project has source files with compilation errors, these files are listed.

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Defects Summary

Create table of defects (Bug Finder only)

## Description

This component creates a table of Polyspace Bug Finder defects. From this table, you can see the number of defects of each type.

## Properties

### Include Checkers with No Defects Detected

If you select this option, the table includes all defect types that Polyspace Bug Finder can detect, including those that do not occur in your code.

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Global Variable Checks

Create table of global variables (Code Prover only)

## Description

This component creates a table of Polyspace Code Prover global variables. From this table, you can see the number of global variables of each type.

## See Also

**Topics**
“Customize Existing Bug Finder Report Template”

# Recursive Functions

Create table of recursive functions

## Description

This component creates a table containing the recursive functions in your source code (along with the files containing the functions).

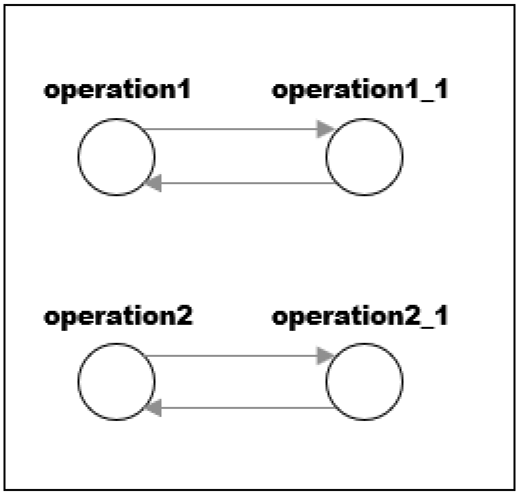
- For each direct recursion (function calling itself directly), the table lists the recursive function.
- For each indirect recursion cycle (function calling itself through other functions), the table lists one function in the cycle.

For instance, the following code contains two indirect recursion cycles.

```
volatile int signal;

void operation1() {
    int stop = signal%2;
    if(!stop)
        operation1_1();
}


void operation1_1() {
    operation1();
}

void operation2() {
    int stop = signal%2;
    if(!stop)
        operation2_1();
}

void operation2_1() {
    operation2();
}

void main(){
    operation1();
    operation2();
}
```

The two call graph cycles are:

- `operation1` → `operation1_1` → `operation1`
- `operation2` → `operation2_1` → `operation2`



This report component shows one function from each of the two cycles: `operation1` and `operation2`. To see the full cycle, open the results in the Polyspace user interface.

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Report Customization (Filtering)

Create filters that apply to your Polyspace reports

## Description

This component allows you to filter unwanted information from existing Polyspace report templates. To apply global filters, place this component immediately below the node representing the report name.

## Properties

### Code Metrics Filters

The properties in table below apply to the inclusion of code metrics in your report.

| Property | Purpose | User Action |
|---|---|---|
| **Include Project Metrics** | Choose whether to include metrics about your Polyspace project. | Select the check box to include project metrics. |
| **Project metrics to include** | Specify project metrics to include or exclude from report. | Enter a MATLAB regular expression. |
| **Include File Metrics** | Choose whether to include per file metrics in report. | Select the check box to include per file metrics. |
| **File Metrics > Files to include** | Specify files to include or exclude when reporting file metrics. | Enter a MATLAB regular expression. |
| **File metrics to include** | Specify file metrics to include or exclude from report. | Enter a MATLAB regular expression. |
| **Include Function Metrics** | Choose whether to include per function metrics in report. | Select the check box to include per function metrics. |
| **Function Metrics > Files to include** | Specify files to include or exclude when reporting function metrics. | Enter a MATLAB regular expression. |
| **Functions to include** | Specify functions to include or exclude when reporting function metrics. | Enter a MATLAB regular expression. |
| **Function metrics to include** | Specify function metrics to include or exclude from report. | Enter a MATLAB regular expression. |

### Coding Rules Filters

The properties in table below apply to the inclusion of coding rule violations in your report.

| Property | Purpose | User Action |
|---|---|---|
| **Files to include** | Specify files to include or exclude when reporting coding rule violations. | Enter a MATLAB regular expression. |
| **Coding rule numbers to include** | Specify coding rules to include or exclude when reporting coding rule violations. | Enter a MATLAB regular expression. |
| **Classifications to include** | Specify classifications to include or exclude when reporting coding rule violations. | Enter a MATLAB regular expression. |
| **Status types to include** | Specify statuses to include or exclude when reporting coding rule violations. | Enter a MATLAB regular expression. |

#### Run-time Check Filters

The properties in table below apply to the inclusion of Polyspace Code Prover checks in your report.

| Property | Purpose |
|---|---|
| **Red Checks** | Specify whether to include red checks in your report. Red checks indicate proven run-time errors. |
| **Gray Checks** | Specify whether to include gray checks in your report. Gray checks indicate unreachable code. |
| **Orange Checks** | Specify whether to include orange checks in your report. Orange checks indicate possible run-time errors. |
| **Green Checks** | Specify whether to include green checks in your report. Green checks indicate that an operation does not contain a specific run-time error. |
| **Inspection Point Checks** | Specify whether to include inspection point checks in your report. These checks allow an user to find the values that a variable can take at a certain point in the code. |
| **Unreachable Functions** | Specify whether to include unreachable functions in your report. |

#### Advanced Filters

The properties in table below apply to the inclusion of metrics, coding rule violations and Polyspace Code Prover checks in your report.

| Property | Purpose | User Action |
|---|---|---|
| **Justification status** | Choose whether to report only justified checks, only unjustified checks or all checks. | Choose an option from the dropdown list. |

| Property | Purpose | User Action |
|---|---|---|
| **Files to include** | Specify files to include or exclude from your report. | Enter a MATLAB regular expression. |
| **Check types to include** | Specify Polyspace Code Prover checks to include in your report. | Enter a MATLAB regular expression. |
| **Function names to include** | Specify functions to include or exclude from your report. | Enter a MATLAB regular expression. |
| **Classification types to include** | Specify classifications to include or exclude from your report. | Enter a MATLAB regular expression. |
| **Status types to include** | Specify statuses to include or exclude from your report. | Enter a MATLAB regular expression. |
| **Comments to include** | Specify comments to include or exclude from your report. | Enter a MATLAB regular expression. |

## See Also

### Topics

"Customize Existing Bug Finder Report Template"
"Regular Expressions"

# Run-time Checks Details Ordered by Color/File

Create overrides for global filters in Polyspace reports (Code Prover only)

## Description

This component adds detailed information about the run-time checks to your report. This component can also be used to override global filters in specific chapters of your report. Use the following workflow when using filters in your report:

1. To create filters that apply to all chapters of your report, use the **Report Customization (Filtering)** component. For more information, see `Report Customization (Filtering)`.
2. To override some of the filters in individual chapters, use the **Run-time Checks Details Ordered by Color/File** component. Select the **Override Global Report filter** box.

## Properties

### Categories To Include

The properties in table below apply to the inclusion of Polyspace Code Prover checks in your report.

| Property | Purpose |
|---|---|
| **Red Checks** | Specify whether to include red checks in your report. Red checks indicate proven run-time errors. |
| **Gray Checks** | Specify whether to include gray checks in your report. Gray checks indicate unreachable code. |
| **Orange Checks** | Specify whether to include orange checks in your report. Orange checks indicate possible run-time errors. |
| **Green Checks** | Specify whether to include green checks in your report. Green checks indicate that an operation does not contain a specific run-time error. |
| **Inspection Point Checks** | Specify whether to include inspection point checks in your report. These checks allow an user to find the values that a variable can take at a certain point in the code. |
| **Unreachable Functions** | Specify whether to include unreachable functions in your report. |

### Advanced Filters

The properties in table below apply to the inclusion of metrics, coding rule violations and Polyspace Code Prover checks in your report.

| Property | Purpose | User Action |
|---|---|---|
| **Justification status** | Choose whether to report only justified checks, only unjustified checks or all checks. | Choose an option from the dropdown list. |
| **Files to include** | Specify files to include or exclude from your report. | Enter a regular MATLAB expression. |
| **Check types to include** | Specify Polyspace Code Prover checks to include in your report. | Enter a regular MATLAB expression. |
| **Function names to include** | Specify functions to include or exclude from your report. | Enter a regular MATLAB expression. |
| **Classification types to include** | Specify classifications to include or exclude from your report. | Enter a regular MATLAB expression. |
| **Status types to include** | Specify statuses to include or exclude from your report. | Enter a regular MATLAB expression. |
| **Comments to include** | Specify comments to include or exclude from your report. | Enter a regular MATLAB expression. |

## See Also

**Topics**

"Customize Existing Bug Finder Report Template"

# Run-time Checks Details Ordered by Review Information

Create table with run-time checks ordered by review information (Code Prover only)

## Description

This component creates tables displaying the Polyspace Code Prover checks in your code. All checks with same combination of **Severity** and **Status** appear in the same table.

## See Also

### Topics

"Customize Existing Bug Finder Report Template"

# Run-time Checks Summary Ordered by File

Create table with run-time checks ordered by file (Code Prover only)

## Description

This component creates a table displaying the number of Polyspace Code Prover checks per file in your code.

## Properties

### Sort the data

Use this option to sort the rows in the table alphabetically by filename or by percentage of unproven code.

### Display as

Use this option to display the number of checks in a table or in bar charts.

### Display ratio of checks in a file

Select this option to display the number of checks of a certain color as a ratio of total number of checks in the file.

### Include checks from Polyspace standard library stub functions

Select this option to include the checks from Polyspace standard library stub functions in your display.

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Software Quality Objectives - Coding Rules Summary

Create table of coding rule violations in results downloaded from Polyspace Metrics

## Description

This component creates a table containing coding rule violations in results downloaded from Polyspace Metrics.

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Software Quality Objectives - Run-time Checks Details

Create table of result details for results downloaded from Polyspace Metrics

## Description

This component creates tables showing results downloaded from Polyspace Metrics.

The component `Software Quality Objectives - Run-time Checks Summary` shows the distribution of results. This component shows individual instances of results. Each file has a dedicated table showing the findings in the file.

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Software Quality Objectives - Run-time Checks Summary

Create table of results summary for results downloaded from Polyspace Metrics

## Description

This component creates a table showing the distribution of run-time checks in results downloaded from Polyspace Metrics.

This component shows the distribution of run-time checks. The component `Software Quality Objectives - Run-time Checks Details` shows the individual instances of run-time checks.

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Summary By File

Create table showing summary of Polyspace results by file

## Description

This component creates a table showing a breakdown of Polyspace results by file.

## See Also

**Topics**
"Customize Existing Bug Finder Report Template"

# Variable Access

Create table showing global variable access in source code (Code Prover only)

## Description

This component creates a table showing the global variable access in your source code. For each global variable, the table displays the following information:

- Variable name.

  The entry for each variable is denoted by |.
- Type of the variable.
- Number of read and write operations on the variable.
- Details of read and write operations. For each read or write operation, the table displays the following information:
  - File and function containing the operation in the form *`file_name.function_name`*.

    The entry for each read or write operation is denoted by `||`. Write operations are denoted by < and read operations by >.
  - Line and column number of the operation.

This table captures the information available on the **Variable Access** pane in the Polyspace user interface.

The table showing variable access contains only the names of files. Below this table, a second table shows the full paths to files (in two columns, **Filename** and **Full filename**). If a variable access occurs in a Standard library function, the two columns contain this information:

- **Filename**: `__polyspace__stdstubs.c` (the file containing Polyspace implementation of Standard Library functions)
- **Full filename**: `Std library`

## See Also

### Topics

"Customize Existing Bug Finder Report Template"

# Variable Checks Details Ordered By Review Information

Create table with global variable results ordered by review information (Code Prover only)

## Description

This component creates tables displaying the Polyspace Code Prover global variable results in your code. All checks with same combination of **Severity** and **Status** appear in the same table.

## See Also

**Topics**
“Customize Existing Bug Finder Report Template”

# Polyspace Bug Finder Assumptions

# 32

# Approximations Used During Bug Finder Analysis

# Inputs in Polyspace Bug Finder

A Bug Finder analysis by default does not return a defect caused by a special value of an unknown input unless the input is subsequently bounded in the code. Polyspace makes no assumption about the value of unbounded inputs when your source code is incomplete. For example, in the following code Bug Finder detects a **division by zero** in `foo_1()`, but not in `foo_2()`:

```
int foo_1(int p)
{
  int x = 0;
  if ( p > -10 && p < 10 )  /* p is bounded by if statement */
      x = 100/p;    /* Division by zero detected */

  return x;
}

int foo_2(int p)    /* p is unbounded */
{
  int x = 0;
  x = 100/p;    /* Division by zero not detected */


  return x;
}
```

To set bounds on your input, add constraints in your code such as `assert` or `if`. At the cost of a possibly longer runtime, you can perform a more exhaustive analysis where all values of function inputs are considered when showing defects. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## See Also

“Global Variables in Polyspace Bug Finder” on page 32-3 | “Bug Finder Analysis Assumptions”

# Global Variables in Polyspace Bug Finder

When you run a Bug Finder analysis, Polyspace makes certain assumptions about the initialization of global variables. These assumptions depend on how you declare and define global variables. For example, in this code

```
int foo(void) {
    return 1/gvar;
}
```

Bug Finder detects a **division by zero** defect with the variable `gvar` in these cases:

- You define `int gvar;` in the source code and provide a `main` function that calls `foo`. Bug Finder follows ANSI standards that state the variable is initialized to zero.
- You define `int gvar;` or declare `extern int gvar;` in the source code. Another function calls `foo` and sets `gvar=0`. Otherwise, when your source files are incomplete and do not contain a `main` function, Bug Finder makes no assumption about the initialization of `gvar`.
- You declare `const int gvar;`. Bug Finder assumes `gvar` is initialized to zero due to the `const` keyword.

At the cost of a possibly longer runtime, you can perform a more exhaustive analysis where all values are considered for each read of a global variable by `foo` or of its callees when showing defects. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values”.

## See Also

“Inputs in Polyspace Bug Finder” on page 32-2 | “Bug Finder Analysis Assumptions”

# Volatile Variables in Polyspace Bug Finder

You use the `volatile` keyword to inform the compiler that the value of a variable might change at any time without an explicit write operation. When you run an analysis, Polyspace Bug Finder makes these assumptions about volatile variables:

- **Global volatile variables**

  - If you declare a global volatile variable as `const`, Polyspace uses the initialization value of the variable or the initialization range if you use the `PERMANENT` **Init Mode** to constrain the range of the variable externally. Polyspace uses the initialization value or range for every read of the variable. See “External Constraints for Polyspace Analysis”.

    For instance, in this code:

    ```
    const volatile volatile_var; // Global variable initialized to 0
    const volatile volatile_var_10=10;
    const volatile volatile_var_drs=3; // Variable constrained to range [-5 .. 5]

    int func(void){
        int i= 10 % volatile_var; // Defect
        int j= 10 % volatile_var_10; // No defect
        int k= 10 % volatile_var_drs; // Defect
        return i+j+k;
    }
    ```

    Polyspace detects an **Integer division by zero** defect for `volatile_var` since it is initialized to zero. Polyspace detects an **Integer division by zero** for `volatile_var_drs` because it is externally constrained to the range [-5 .. 5]. All reads of `volatile_var_10` cause no defect.

  - For non-`const` global volatile variables, Polyspace ignores the initialization value of the variable, and then considers the input unknown for each read of the variable. If you use the `PERMANENT` **Init Mode** to constrain the range of the variable externally, Polyspace uses this range for every read of the variable. See “External Constraints for Polyspace Analysis”.

    For instance, in this code:

    ```
    volatile volatile_var; // Global variable initialized to 0
    volatile volatile_var_drs=3; // Variable constrained to range [-5 .. 5]

    int func(void){
        int i= 10 % volatile_var; // No defect
        int j= 10 % volatile_var_drs; // Defect
        return i+j;
    }
    ```

  Polyspace detects an **Integer division by zero** defect for `volatile_var_drs` because it is externally constrained to the range [-5 .. 5]. All reads of `volatile_var` cause no defect.

- **Local volatile variables**

  Polyspace ignores the initialization value of local volatile variables, and then considers the input unknown for each read of the variable. For example, in this code:

  ```
  int foo(void){
      volatile var=0;
      return 1/var; // No defect
  }
  ```

  Polyspace detects no defect. You cannot use external constraints to constrain the range of local variables.

At the cost of a possibly longer runtime, you can perform a more exhaustive analysis where Polyspace considers all values for each read of a volatile variable. See `Run stricter checks considering` `all values of system inputs (-checks-using-system-input-values)`. When you use this option to analyze all the preceding code examples, Polyspace detects additional **Integer division by zero** defects on the lines labeled with comment `// No defect`, including for the local volatile variable example.

## See Also

"Inputs in Polyspace Bug Finder" on page 32-2 | "Bug Finder Analysis Assumptions"

# Duplicate Code Detection in Polyspace Bug Finder

Sections of code that do the same operations require unnecessary additional maintenance. Duplicated code also increases the chances you will update the code in one place but forget to update the other. Using Polyspace Bug Finder, you can detect sections of code that are exact or partial duplicates of each other.

## Duplicate Code Checkers

Polyspace Bug Finder reports duplicate code using these checkers. Using these checkers, you can find opportunities for refactoring your code into more maintainable modules.

| Defect | Description | Defect Group |
|---|---|---|
| `Duplicated code` | A section of code is duplicated in other places. | Good Practice |
| `Partially duplicated code` | A section of code is duplicated in other places with very minor changes. | Good Practice |
| `Possible copy-paste error` | A section of code is duplicated in other places with exactly one minor change. | Programming |

## Limitations of Duplicate Code Detection

The defect checker does not flag certain blocks as duplicates. For instance:

- The blocks of code considered duplicates have to typically consist of more than six consecutive lines. This restriction is put in place to prevent excess false positives. For instance, if a single line or a couple of lines is repeated in multiple places, this repetition is not an issue.
- The checker does not detect duplicate code sections containing the following constructs:

  - Conditional statements – `if`, `switch`, and so on.
  - Loops – `for`, `while`, and so on.

  The checker detects only those duplicate code sections that do not involve branching.
- The checker resolves preprocessor directives such as `#ifdef`-s before analyzing the code for duplicate blocks. For instance, depending on whether the macro `PCDEBUG` is defined or not, the lines in the following `#ifdef` form part of the analysis or get removed from the analysis:

  ```
  #ifdef PCDEBUG
  //Block of code
  #endif
  ```